MASTER EN SCIENCES INFORMATIQUES

Sur l'optimisation de programmes écrits dans des langages algébriques possédant les propriétés de séquentialité et d'affectation

Van Cauwenbergh, J.

Award date:
1974

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix - Namur

INSTITUT D'INFORMATIQUE

1973-1974

Sur l'optimisation de programmes écrits dans des langages algébriques possédant les propriétés de séquentialité et d'affectation

J. VAN CAUWENBERGH

MÉMOIRE

présenté pour l'obtention
du grade de
Licencié et Maître
en Informatique

A mes parents

REMERCIEMENTS

Que l'ensemble du corps professoral de l'Institut d'Informatique trouve ici l'expression de nos remerciements pour l'enseignement qu'ils nous ont prodigué.

Toute notre gratitude va à Monsieur J. Fichefet, pour ses conseils judicieux et l'aide constante qu'il nous a apportée lors de la préparation et de la rédaction de ce mémoire.

Nous tenons à remercier bien vivement le professeur D.C. Cooper, professeur au "University College of London", pour les nombreuses et intéressantes discussions que nous avons eues avec lui et pour l'intérêt particulier qu'il a porté à notre travail.

Nous exprimons toute notre reconnaissance au staff du"Computer Science Department of the University College of Swansea" et plus particulièrement à Messieurs R.D. Dowsing et M.G. Farringdon, pour les discussions chaleureuses que nous avons eues avec eux, ainsi que pour l'aide qu'ils nous ont apportée.

Une pensée toute spéciale va à Madame Noël qui assura avec art l'achèvement matériel de ce mémoire.

PROCEDURE POUR LA LECTURE DE CE MEMOIRE

1) Lisez l'introduction.

2) Si Vous êtes intéressé par l'optimisation de programme
   Alors Allez en 3
   Sinon Refermez ce mémoire.

3) Lisez le chapitre 1 et, si vous avez du courage, analysez les exemples.

4) Si Vous êtes uniquement intéressé par l'optimisation orientée segment d'instructions
   Alors Lisez le chapitre 2 et si vous désirez apprendre un nouveau langage, lisez le paragraphe 2.1.9. Allez en 5.
   Sinon Allez en 7

5) Etes-vous courageux et le chapitre 2 vous a-t-il intéressé ?
   Si Oui Alors Lisez aussi le chapitre 3 avant de refermer ce mémoire. Si vous êtes de plus en plus intéressé par ce travail, continuez sa lecture.

6) Lisez la conclusion et allez en 8

7) Etes-vous intéressé à la fois par l'optimisation orientée segment d'instructions et l'optimisation plus globale ?
   Si Oui Alors Lisez l'ensemble de ce mémoire et reposez-vous de temps en temps. Allez en 6.
   Si Non Alors Vous vous intéressez uniquement à l'optimisation décrite dans les chapitres 3, 4, 5, 6, 7. Lisez ces chapitres et allez en 6.

8) Si 1 + 1 = 3
   Alors Refermez ce mémoire
   Sinon Lisez les annexes 1 et 2.

9) Dans les conclusions de ce mémoire, vous avez trouvé quelques sujets que nous n'avons pas étudiés.A vous de les développer.

Nous vous remercions pour la lecture de ce mémoire.

— § —

TABLE DES MATIERES

——— § ———

INTRODUCTION

1. Historique

Très tôt, on s'est aperçu que l'on pouvait améliorer le code objet résultant d'une compilation. Les premiers articles relatifs à l'optimisation de code ne traitent toutefois que des éléments particuliers d'un langage algébrique.

Les chercheurs ont tout d'abord localisé leur attention sur l'amélioration du code objet résultat de la compilation d'expressions arithmétiques, notamment sur la limitation du nombre de cases temporaires utilisées dans le code, la réduction du nombre d'opérations de "Load" et "Store", ...

L'article de Floyd [FLOYD 61] est sans doute le premier qui envisage la propagation de constantes et l'élimination de sous-expressions communes.

Avec la création de langages ayant la puissance d'Algol 60, des algorithmes pour la compilation efficiente des expressions booléennes ont été développés.

Le premier article traitant vraiment de l'optimisation de programme est celui d'Allen [ALLEN 69]. Cet article a servi de point de départ pour notre travail.

Lowry et Medlock [LM 69] ont réalisé le premier optimiseur commercial (compilateur IBM Fortran H). Dans cet optimiseur, ils ont tenu compte des aspects d'optimisation envisagés ici.
Depuis lors, des recherches théoriques sur l'optimisation "constructive" de programme ont été entamées par Allen, Cocke et Schwartz [
Elles généralisent certaines techniques d'optimisation abordées ici mais sont encore assez fragmentaires. Parallèlement, d... recherches théoriques sur la formalisation de la notion même d'optimisation de programme - elles portent entre autres sur les notions d'équivalence de programmes, process calculus, transformations de programmes - sont effectuées mais sont restreintes à des formes très particulières de programmes ("straight line code") [AHO 70], [BALLARD 72].

2. Point de vue adopté dans ce mémoire

Les techniques d'optimisation de programme peuvent être classifiées en deux catégories distinctes: l'optimisation locale et l'optimisation globale.

L'optimisation locale consiste en l'application au programme lors de la génération de code, de techniques utilisant les caractéristiques et particularités de la machine pour laquelle le compilateur est écrit. Il s'agit d'un ensemble de "trucs" (tricks) tels que shifts à gauche au lieu de multiplication par 2, ... Nous n'envisagerons pas ici ce type d'optimisation pour deux raisons principales :

1° elle nous obligerait à considérer plusieurs machines et à énoncer l'ensemble des "trucs" pour chacune d'elles;

2° elle ne semble pas pouvoir donner naissance à une théorie.

C'est l'optimisation globale qui retiendra notre attention dans ce travail. Nous en donnerons une définition dans le paragraphe 3.
Remarquons encore que l'optimisation de programme poursuit deux buts, à savoir l'optimisation de l'espace mémoire et la réduction du temps d'exécution des programmes.
Nous nous sommes limités ici à l'optimisation produisant une réduction du temps d'exécution des programmes. Actuellement, peu de travaux traitent le problème de la réduction de l'espace mémoire (/ERSHOV 67/, /NIEVERGLET 65/). Nous pensons que ce problème peut être résolu au niveau des "operating systems" (mémoire virtuelle).

En raison de leur aspect embryonnaire, nous n'aborderons pas ici les recherches actuelles d'Allen, Cocke et Schwartz déjà signalées ci-dessus. Nous proposons cependant dans les annexes de ce mémoire, un algorithme théorique analysant le flot des données dans un programme. Il nous a semblé préférable de suivre le chemin des écoliers afin de mieux comprendre les problèmes posés par l'optimisation de programme. Cette démarche nous laisse la porte ouverte pour des recherches futures.

Nous n'aborderons pas non plus l'aspect relatif à la formalisation de l'optimisation de programme. Il nous semble qu'avant de le faire, nous devons avant tout étudier les travaux concernant l'équivalence de programmes,

la correction des programmes, la formalisation des notions de programme et d'exécution de programme. Nous avons néanmoins jeté un coup d'oeil rapide sur ces notions sans les approfondir. Il nous a semblé préférable d'étudier d'abord "intuitivement" l'optimisation de programme afin de mieux comprendre ses mécanismes.

3. Définition de l'environnement(*)

Il est naturel de se poser deux questions essentielles à propos de l'optimisation de programme. La première: pourquoi optimiser et dans quel but ? La seconde: jusqu'où peut-on aller ?

Nous répondrons d'abord à la seconde question en disant que le problème de l'équivalence de programmes, en toute généralité, est indécidable. Ceci implique que l'on ne pourra pas trouver de techniques pouvant produire le programme le plus optimal. Il est cependant possible de trouver des techniques améliorant sensiblement un programme et même une théorie produisant une classe de programmes équivalents et optimisés.

Avant de répondre à la première question, nous donnerons une description du problème de l'optimisation.

Dans ce travail, nous considérons l'ensemble $\mathcal{L}$ des langages algébriques possédant entre autres les propriétés de séquentialité et d'affectation de valeur.

La sémantique d'un langage L décrit en quelque sorte une machine abstraite M_L qui exécutera tous les programmes écrits dans le langage L et syntaxiquement corrects. Le compilateur C_L pour le langage L est en quelque sorte un interface entre M_L et la machine réelle. Un interpréteur I_L pour ce langage L est comparable à un simulateur de la machine M_L.

La propriété de séquentialité d'un langage L peut s'énoncer de la façon suivante :

"Lorsqu'une instruction i appartenant à un programme P écrit dans le langage L a été exécutée par la machine M_L, celle-ci connaît, implicitement

---

(*) Dans ce paragraphe, optimisation est synonyme d'optimisation globale.

ou par une indication explicite dans P, l'instruction i' suivante à exécuter".

La propriété d'affectation de valeur d'un langage L peut s'énoncer de la façon suivante :

"Un programme P écrit dans le langage L modifie, lors de son exécution par M_L, le contenu de la mémoire de M_L. Pour ce faire, P utilise un système d'"adressage de la mémoire de M_L".

Nous ne considérons que des langages ayant ces propriétés car :

- il est possible d'associer un graphe à tout programme écrit dans un langage possédant la propriété de séquentialité. Le formalisme et certaines méthodes de la théorie des graphes sont dès lors utilisables. La récolte d'informations globales et les possibilités de manipulation globale des programmes sont ainsi facilitées;
- la propriété d'affectation de valeur permet d'ordonner les actions du programme.

Etant donné un langage $L \in \mathcal{L}$ et un programme P écrit dans ce langage, l'optimisation de programme consiste en l'application à P d'une combinaison de techniques globales en vue d'associer à P un autre programme P' plus optimal. L'expression "combinaison de techniques" désigne l'application à P de plusieurs techniques différentes mais exécutées dans un certain ordre. Le programme P' doit être équivalent, en un certain sens, au programme P. Il nous semble donc utile de discuter de l'équivalence de programmes avant de définir la notion d'optimalité.

Il existe un grand nombre de définitions d'équivalence de programmes menant toutes vers des problèmes indécidables. Dès lors, nous ne définirons pas explicitement l'équivalence de programmes. Nous donnerons cependant une idée intuitive du type de définition qu'il faudrait chercher.

Nous nous contenterons ici de la définition suivant laquelle deux programmes P et P' sont équivalents si, et seulement si, pour un ensemble de données et pour une même machine réelle, P et P' délivrent des résultats identiques ou égaux à un ε près, quelle que soit la forme sous laquelle les données et les résultats sont présentés.

Avant d'énoncer un principe général d'optimalité, considérons quelques cas triviaux.

- Soient P et P' deux programmes équivalents et possédant les mêmes temps d'exécution: le programme possédant le plus petit volume sera le plus optimal.
- Soient P et P' deux programmes équivalents et possédant des volumes égaux: celui des deux conduisant au plus petit temps d'exécution sera le plus optimal.

Ces deux exemples suffisent pour montrer que la notion d'optimalité dépend fortement des objectifs que l'on se fixe. Pour définir cette notion de façon générale, il faut trouver un système qui, à un programme P, associe un coût C(P) en tenant compte du temps d'exécution et du volume du programme P. De cette façon, si P et P' sont deux programmes équivalents et si $C(P) \geqslant C(P')$, alors P' est au moins aussi optimal que P.

Les notions introduites ci-dessus sont généralisables à plusieurs programmes.
Lorsqu'on fait de l'optimisation de programme, on poursuit donc deux objectifs :

1° conserver l'équivalence entre programmes;

2° obtenir un programme plus optimal.

Nous n'avons pas une connaissance explicite de la fonction de coût C(P). Il nous semble cependant raisonnable de supposer que les techniques envisagées ci-après :

- propagation de constantes (chapitre II),
- élimination d'instructions redondantes (chapitre II),
- déplacement d'instructions invariantes (chapitre IV),
- réduction des opérateurs (chapitre V),
- remplacement de tests (chapitre VI),
- élimination des assignations mortes (chapitre VI),

auront pour effet de diminuer C(P). Cela a d'ailleurs été vérifié empiriquement par Knuth [KNUTH 71] pour des programmes écrits en Fortran 4.

Ceci répond en partie à la première question posée en début de ce paragraphe. Nous en discuterons à nouveau dans les conclusions de ce mémoire, à la lumière des techniques qui sont proposées.

On peut aussi se demander s'il est utile d'envisager un système optimiseur indépendamment d'un compilateur.
Pour compiler un programme, un compilateur n'utilise généralement qu'une information locale. Mais pour faire de l'optimisation de programme, il faut une information globale sur le programme. Les procédures nécessaires à la collecte de cette information sont suffisamment complexes que pour les isoler en un système indépendant.

Comment peut-on concevoir un tel système ? Plus particulièrement, à quel niveau va-t-on le placer ?

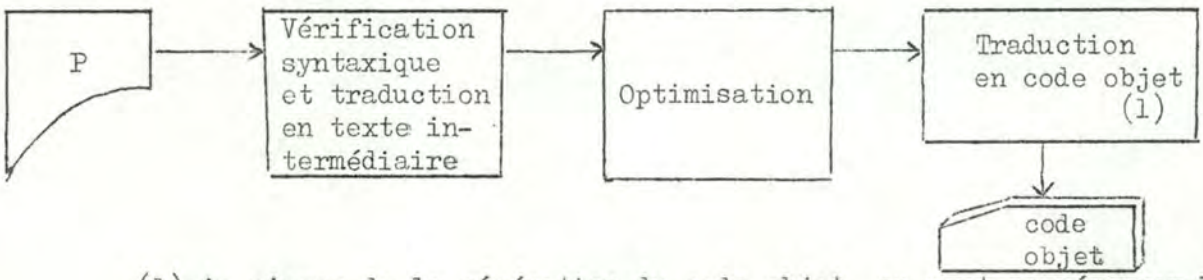
Un premier système possible consiste à optimiser au niveau du programme source: un tel système optimiseur traiterait un programme P écrit dans un langage source L et donnerait comme résultat un programme P' écrit dans L mais plus optimal que P.
Cette façon de procéder présente certains inconvénients :

1° pour réaliser une élimination efficiente d'instructions redondantes, il faut passer par une forme intermédiaire [SCHNECK 72],

2° il est impossible, par un tel système, d'optimiser le calcul d'adresse qu'implique l'utilisation de tableaux à plusieurs dimensions,

3° la réduction des opérateurs est moins efficiente dans un tel système.

Un deuxième système possible est l'optimisation au niveau du code intermédiaire: l'optimisation de programme se situe donc après les phases d'analyse syntaxique et de traduction en forme intermédiaire. C'est ce système que nous avons adopté dans ce travail. Il permet d'éviter les inconvénients du premier système et présente, comme nous le verrons, de nombreux autres avantages. Il implique le schéma de travail suivant :



(1) Au niveau de la génération du code objet, on peut se réserver la possibilité d'insérer un système d'optimisation locale.

Notons encore qu'en raison de la complexité croissante des ordinateurs, des langages et des systèmes, il est intéressant de chercher une réponse aux question suivantes.

1° Des programmes éctits en langage évolué peuvent-ils être compilés (via un compilateur-optimiseur) pour donner un code objet compétitif avec les mêmes programmes écrits en langage d'assemblage ?

2° Peut-on écrire un système (O.S., compilateur) en langage évolué sans dégrader les performances du système ?

3° Si la réponse aux deux premières questions est positive, peut-on minimiser le temps de compilation ?

Nous répondrons à ces questions dans les conclusions de ce travail.

—— § ——

Chapitre I

DEFINITION DES CONCEPTS DE BASE ET REMARQUES SUR LEUR CONSTRUCTION A PARTIR D'UN LANGAGE SOURCE DONNE

Un système automatique d'optimisation utilise une information globale sur le programme afin d'améliorer le programme source et ceci à l'insu du programmeur.

Le programme source est un formalisme représentant une codification plus ou moins parfaite d'un algorithme, d'un calcul. Comme on désire améliorer, sans toutefois le changer, l'algorithme codifié, il est nécessaire de déterminer ce que représente la codification.

Le texte intermédiaire nous permettra de représenter le "sens" * de la codification source.

Un algorithme pouvant être considéré comme un ensemble d'actions qu'il faut exécuter dans un certain ordre,il est essentiel de retrouver les actions et l'ordre, c'est-à-dire la séquence.

1. La séquence

La séquence des actions composant l'algorithme ne saurait être mieux décrite que par l'organigramme du programme. Nous voyons donc naître l'idée d'associer au programme source un graphe orienté que nous appellerons diagramme des flots (d'après l'anglais "Flow-diagram" ou "Flow-chart").

Le graphe que nous associerons au programme source n'aura pas comme but unique de représenter le caractère d'ordre des actions, mais, comme nous le verrons plus loin, il permettra l'obtention de la réponse à certaines questions posées par les différentes techniques d'optimisation, ceci sans trop d'analyse et en appliquant des procédures connues (celles de la théorie des graphes).

---

* Sens n'est pas le terme adéquat. Nous l'utilisons néanmoins car le texte intermédiaire est généralement produit par les routines sémantiques du compilateur.

2. Les actions

Il reste à trouver un système permettant de représenter les actions composant l'algorithme, un formalisme dans lequel un traducteur pourra traduire la "sémantique" du programme source P.

Suivant la "finesse" du traitement que l'on désire réaliser, ce formalisme est plus ou moins riche. Ainsi, si nous désirons réaliser un compilateur optimiseur Algol 60, il est intéressant que le formalisme permette l'expression de la "sémantique" de la structure de bloc Algol 60 et des particularités de l'appel par nom ou par valeur.
Le formalisme doit de plus posséder cette qualité non négligeable qu'est la simplicité. Il doit permettre l'expression, en des opérations simples, de la "sémantique" d'instructions sources complexes. Un exemple rendra cette affirmation plus claire. Prenons l'expression arithmétique FORTRAN 4 :
P * K * (K + 2.0)/(K + 1) * * 2. Elle sera évaluée de la façon suivante :

$$
\begin{array}{rcll}
K + 1 & \longrightarrow & Z_1 & \text{(Integer)} \\
Z_1 \ast\ast 2 & \longrightarrow & Z_2 & \text{(Integer)} \\
Z_2 & \longrightarrow & Z_3 & \text{(Integer} \longrightarrow \text{Real)} \\
K & \longrightarrow & Z_4 & \text{(Integer} \longrightarrow \text{Real)} \\
Z_4 + 2.0 & \longrightarrow & Z_5 & \text{(Real)} \\
K & \longrightarrow & Z_6 & \text{(Integer} \longrightarrow \text{Real)} \\
P \ast Z_6 & \longrightarrow & Z_7 & \text{(Real)} \\
Z_7 \ast Z_5 & \longrightarrow & Z_8 & \text{(Real)} \\
Z_8 / Z_3 & \longrightarrow & Z_9 & \text{(Real)}
\end{array}
$$

Nous nous trouvons en face de neuf actions ou opérations résumées par une seule expression du langage FORTRAN 4. Remarquons que deux opérations $K \longrightarrow Z_4$ et $K \longrightarrow Z_6$ sont identiques. Par le fait de la décomposition en actions élémentaires, nous avons pu mettre en évidence cette redondance.

Pour des raisons d'indépendance, il est souhaitable que le formalisme soit dégagé des caractéristiques propres au langage évolué dans lequel est codifié l'algorithme. De même, il est souhaitable qu'il soit dégagé des caractéristiques propres au langage de la machine.

1.1. Définition du graphe associé au programme source

Ce graphe $\mathcal{G}$ (*) est entièrement défini par l'ensemble X de ses sommets et l'ensemble U de ses arcs.

1.1.1. Segment d'instructions S. Définition de l'ensemble X des sommets

Un segment d'instructions est une séquence linéaire d'instructions ayant un seul point d'entrée et un seul point de sortie.

Une séquence linéaire d'instructions est une séquence $I_1, \ldots, I_n$ d'instructions telle que si I_j et I_i $(1 \leq j < i \leq n)$ sont deux instructions de la séquence, I_j sera toujours exécutée avant l'instruction I_i.

"Un seul point d'entrée" interdit toute référence externe au segment à une instruction située à l'intérieur du segment. Seule l'instruction I_1 peut être adressée.

"Un seul point de sortie" interdit la présence d'une instruction de branchement à l'intérieur d'un segment. Seule I_n peut être une instruction de branchement.

(*) au sens de König- Berge.

Les segments d'instructions sont construits par les routines sémantiques du compilateur. On trouvera quelques exemples dans 1.2.
L'ensemble X des sommets de G est l'ensemble des segments d'instructions du programme.

Certaines techniques d'optimisation - notamment la suppression d'instructions redondantes - ont le segment d'instructions pour seule donnée de travail et sont souvent fortement limitées du point de vue de leur efficacité en raison de la longueur (en nombre d'instructions) souvent très petite de ce segment d'instructions. Par exemple, Knuth [KNUTH 71] a montré qu'en FORTRAN 4, la longueur moyenne d'un segment d'instructions est de 3 instructions sources.

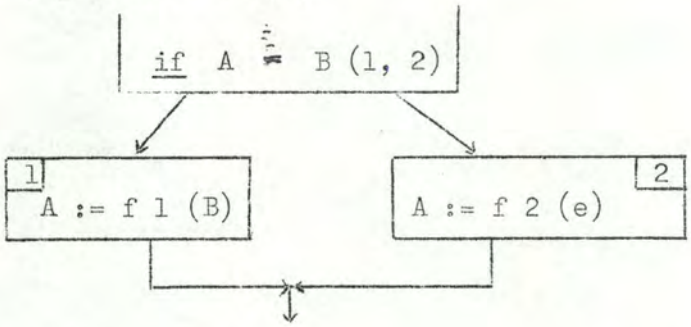
La longueur peu importante des segments d'instructions est due principalement au fait que les programmeurs utilisent souvent des instructions conditionnelles. Un exemple fera mieux comprendre cette remarque. L'instruction ALGOL 60 suivante :

<u>if</u> A = B <u>then</u> A := f 1(B) <u>else</u> A := f 2(e);

ou son équivalent sémantique suivant :

A := <u>if</u> A = B <u>then</u> f 1(B) <u>else</u> f 2 (e);

donnera naissance à 2 segments composés chacun d'une seule instruction, à savoir A := f 1 (B) et A := f 2(e)



1.1.2. <u>Ensemble U des arcs</u>

Nous dirons qu'un arc (S_i, S_j) relie deux segments S_i et S_j si <u>le "contrôle" du programme peut passer du segment S_i au segment S_j</u>.

Par "contrôle" du programme, il faut comprendre "séquence d'exécution" : une fois les instructions du segment S_i exécutées, celles du segment S_j peuvent être exécutées.

Les instructions sources terminant un segment d'instructions sont généralement les instructions de branchement, les instructions conditionnelles, les appels de procédures si ceux-ci ne possèdent pas exclusivement la propriété du "contrôle" implicite.

La propriété du "contrôle" implicite est celle que possède une instruction qui, une fois exécutée, provoque l'exécution de l'instruction figurant derrière elle dans le texte du programme.

Du point de vue du "contrôle", les instructions d'un langage évolué ou de base peuvent généralement être séparées en deux classes: la classe des instructions possédant exclusivement la propriété du "contrôle" implicite et la classe de celles ne possédant pas que cette propriété. Les instructions appartenant à cette dernière classe donnent naissance à un ou plusieurs arcs du graphe .
Remarquons cependant que, dans certains cas - tels que celui de l'instruction A := <u>if</u> A = B <u>then</u> $f_1(B)$ <u>else</u> $f_2(C)$ -,il est difficile de dire qu'une instruction se situe dans l'une ou l'autre classe.

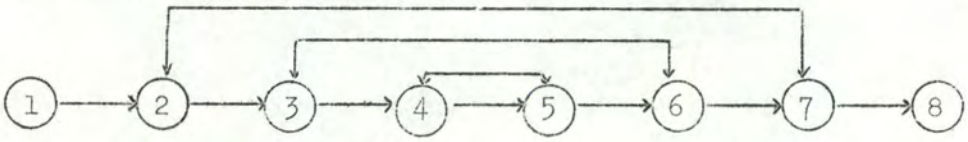
Nous pensons que la façon dont une instruction est traduite en forme intermédiaire détermine son appartenance à l'une des deux classes.

1.1.3. Exemple de programme divisé en segments d'instructions

	Numéro de segment
SUBROUTINE TALLEY (L_1, L_2, L_S)	
DIMENSION LIST 1 (11,50), LIST 2 (10, 100)	1
Q = 0.0	
DO 5 I = 1,L_1	
T = 0.0	
DO 15 J = 1, L_2	2
DO 20 K = 1, L_S	3
IF (LIST 1 (K, I) - LIST 2 (K,J)) 15, 20, 15	4
20 CONTINUE	5
T = T + 1.0	
15 CONTINUE	6

```
LIST 1 (L_S + L, I)  =  T                                  7
Q  =  Q + T
5 CONTINUE
-------------------------------------------------------------------------
RETURN
END                                                        8
-------------------------------------------------------------------------
```

Le graphe associé est le suivant



1.2. "Formalisme" intermédiaire

Le "formalisme intermédiaire est constitué d'un ensemble de tables dont nous donnons une description sommaire. Le problème d'organisation de ces tables ne sera pas considéré dans ce travail; il appartient à chaque "implémenteur" de décider de l'organisation qu'il souhaite. Ce "formalisme" intermédiaire permettra de représenter le "sens" et la séquence d'un programme source.

Notre problème n'est pas l'étude des tables d'un compilateur. Nous donnerons cependant des renseignements permettant de les construire en fonction d'une phase d'optimisation du programme.

1.2.1. Format général de représentation d'un opérande d'une instruction intermédiaire

Il est parfois intéressant de distinguer les différents types d'opérandes (exemple: procédure "Folding") qui se présentent dans une instruction intermédiaire, sans devoir rechercher l'information dans une table. Il est de plus souhaitable que toutes les tables de la forme intermédiaire possèdent un format fixe. Nous proposons les formats suivants pour réaliser ces objectifs :

1.2.1.1. Variables simples

(CODE)	(I)	(POINTER)
2	0/1	γ_i

"Code" est un entier possédant la valeur 2 pour des variables simples

"I" est un bit d'indirection

"γ_i" est la valeur de "pointer", c'est-à-dire une référence à la table des variables simples où l'on trouve une entrée décrivant la variable simple.

1.2.1.2. Constantes

(CODE)	(I)	(POINTER)
1	0	γ_i

"Code" est un entier possédant la valeur 1 pour des constantes

"I" est positionné à zéro pour signaler qu'il n'y a pas d'indirection

"γ_i" est la valeur de "pointer", c'est-à-dire une référence à la table des constantes.

1.2.1.3. Instructions intermédiaires

(CODE)	(I)	(POINTER)
3	0/1	α_i

"Code" possède la valeur 3 pour une instruction intermédiaire qui est opérande d'une autre instruction intermédiaire

"I" est un bit d'indirection

"α_i" est la valeur de "pointer", c'est-à-dire une référence à la table des instructions.

1.2.1.4. Variables indicées

(Code)	(I)	(POINTER)	(CODE)	(I)	(POINTER)
4	0/1	γ_i	1/2/3	0/1	γ_j
		(ARG)			(SUBS)

"γ_i" est la valeur de "Pointer.arg", c'est-à-dire une référence à la table des tableaux où nous trouvons une entrée décrivant le tableau.

La partie "Subs" s'explique d'elle-même en utilisant 1.2.1 (1, 2, 3).

Remarque - Pour évaluer l'adresse d'un élément d'un tableau, il y a toujours dans la formule du calcul d'adresse une partie constante (CONSTANTPART) et une part e variable (VARIABLEPART). Nous supposons que "CONSTANTPART" est calculée à la compilation (ou à l'exécution) mais que son résultat (ou la référence à son résultat) se trouve dans la table des tableaux.

"Pointer-Subs" contient, lui, la référence au résultat du calcul de "VARIABLEPART".

1.2.2. Structure des tables de quantités (variables, constantes, tableaux)

Parmi ces tables, nous trouvons la table des variables simples (TS), la table des constantes (TC) et la table des tableaux (TT). Elles contiennent des informations relatives aux variables simples, constantes ou tableaux utilisés par le programmeur. Une partie de ces informations sera utilisée par l'optimiseur, notamment :

- le type de la quantité,
- l'indication que la quantité est un opérande global d'une procédure et le nom de cette procédure,
- l'indication que la quantité est un paramètre formel d'une procédure, le type de correspondance entre le paramètre formel et le paramètre actuel, le nom de la procédure,
  l'indication que la qu ntité est un tableau.

Le reste des informations n'est pas utilisé par l'optimiseur lui-même, mais par l'environnement dans lequel il se trouve, c'est-à-dire le compilateur. Parmi ces informations, on trouve :

- l'indication que la quantité fait partie d'une liste d'équivalence, un chaînage avec les autres quantités faisant partie de cette liste,
- adresse "run-time" de la quantité,
- "flags" d'erreurs,
- etc...

1.2.2.1. TS

Format général d'une entrée:

γ_i : identificateur (IDEN); [I, , DP, C, L, ...] (ATRIBUT);
[CO, $\overline{CO}$] (GLOBAL); [$\overline{P}$, P] (FORMAL); type p (TYPE COR), reste;

γ_i ($i \in N^+$) est le pointeur référençant dans la forme intermédiaire la variable simple value (identificateur);

identificateur contient l'identificateur source de la variable simple, ou tout identificateur généré par le compilateur;

[I, R, DP, C, L, ...] détermine le type de la variable simple, I pour integer, R pour real, DP pour double précision, C pour complex, L pour logical, etc...

[CO, $\overline{CO}$] détermine le caractère global (CO) ou non ($\overline{CO}$) de la variable simple;

[P, $\overline{P}$] indique si la variable simple est un paramètre formel ou non d'une procédure;

type p détermine le type de correspondance (appel par valeur, par nom, etc..)

reste est l'ensemble des renseignements supplémentaires.

Exemple : real abasladictature; integer vivelajovialité;

γ_i : abasladictature; R; $\overline{CO}$; $\overline{P}$; Δ ; ...;

γ_j : vivelajovialité; I; $\overline{CO}$; $\overline{P}$; Δ ; ... ;

Déclaration en Algol-Jonquille-7 de TS :

[1: AVS] STRUCTURE TS (CHARACTER (20) IDEN, BINARY (4) ATRIBUT, BINARY GLOBAL, BINARY FORMAL, BINARY (3) TYPECOR, ...)

1.2.2.2. TC

Format général d'une entrée :

γ_i : [I, R, L, DP, C, ...] ; CST;

γ_i et [I, R, L, DP, C, ...] ont la même interprétation que dans 1.2.2.1.

CST est la zone qui contiendra la valeur de la constante. Cette valeur se trouve dans un format propice au calcul direct par l'utilisation des instructions du langage machine.

Exemple : constante 25 sur IBM 360.

γ_3 : I; $(00\ 00\ 00\ 19.)_{16}$;

Déclaration en Algol-Jonquille-74

[1 : NCBR] STRUCTURE TC (BINARY (4) ATRIBUT, EXCEPT (STRUCTURE) VAL);

1.2.2.3. TT

Format général d'une entrée :

γ_i : identificateur (IDEN); [I, R, DP, C, L, ...] (ATRIBUT); [CO, $\overline{CO}$] (GLOBAL); [$\overline{P}$, P] (FORMAL); type p (TYPECOR); ND (NUMBOUNDS);

$d_1 : D_1$, $d_2 : D_2$, ..., $d_{ND} : D_{ND}$; reste; ↑

γ_i, identificateur, [I, R, DP, C, L, ...] , [CO, $\overline{CO}$] , [$\overline{P}$, P], type p; s'interprètent de la même façon que dans 1.2.2.1.

ND est le nombre de dimensions du tableau

$(d_i : D_i)$ est le couple "LOWERBOUND", "UPPERBOUND" pour la $i^{ème}$ dimension.

Si le tableau est un paramètre formel, ND a une valeur zéro.

Exemple : real array vivelanarchie [-20 : 10, 30 : 40, 3 : 3]
real array attentionaux
real array conséquences [n : m, $n_1 : m_1$, $n_2 : m_3$]

γ_{10} : vivelanarchie; R; CO; $\overline{P}$; Δ ; 3; -20 : 10, 30 : 40, 3 : 3; ...; ↑
γ_{20} : attentionaux; R; $\overline{CO}$; P; "value"; 0; ...; ↑
γ_{30} : conséquences; R; CO; $\overline{P}$; Δ ; 3; n : m; $n_1 : m_1$, $n_2 : m_3$; ..., ↑

Si le tableau est à dimensions variables (ND variable ou "LOWERBOUND" ou "UPPERBOUND" variable), les zones correspondantes sont bien entendu indéterminées. On y trouve dès ce moment la référence intermédiaire aux quantités définissant ces zones.

Déclaration en Algol-Jonquille-74

[1 : NVT] STRUCTURE TT (CHARACTER (20) IDEN; BINARY (4) ATRIBUT, BINARY FORMAL; BINARY (3) TYPECOR, (INTEGER, REFERENCE) NUMBOUNDS,
[1 : IF INTEGERTEST (NUMBOUNDS) THEN NUMBOUNDS ELSE CONTENTS (NUMBOUNDS)]
STRUCTURE LIMITS ((INTEGER, REFERENCE) LOWERBOUND, (INTEGER, REFERENCE) UPPERBOUND), STRUCTURE RESTE ...)

1.2.3. Table de définition des segments d'instructions TDS

Cette table localise les segments d'instructions dans la table de séquence TSQ (cf. 1.2.4). Elle contient également un dictionnaire du graphe associé au programme source.

Format général d'une entrée :

n; γ_n; PTLS; PTLP; PTPR; IMFR;

1) n est le numéro du segment d'instructions. Les segments sont numérotés d'après leur entrée dans la table TDS.

2) γ_n est un pointeur vers la première instruction du segment dans la table TSQ (cf. 1.2.4).

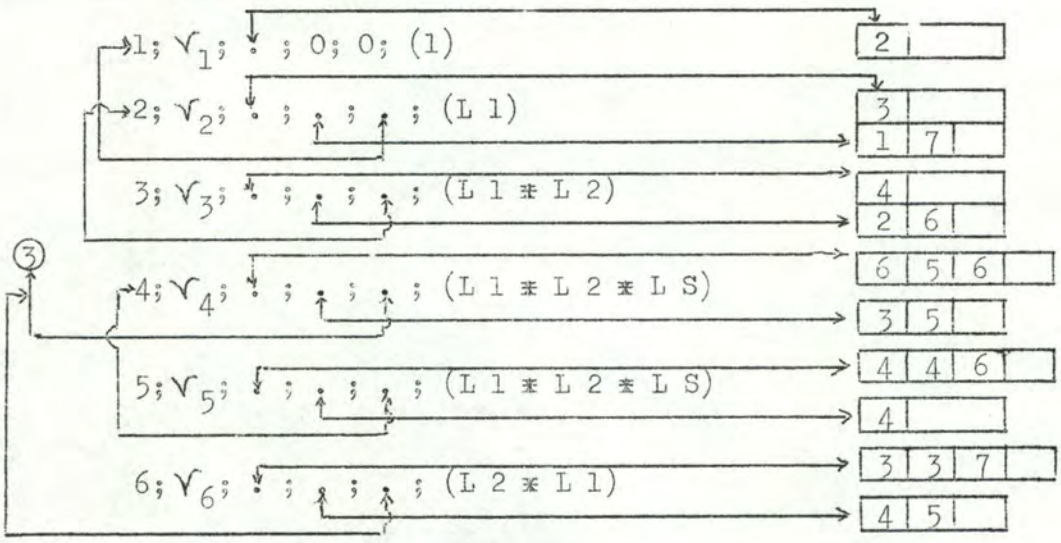
3) PTLS est un pointeur vers la liste des successeurs * du segment dans le graphe. Cette liste est ordonnée suivant un certain ordre (1.2.5.3), elle est exprimée en terme de numéros de segments.

4) PTLP : voir PTLS, mais ici il s'agit des prédécesseurs * du segment.

5) PTPR est un pointeur vers l'entrée de la table TDS qui représente le prédominateur* immédiat du segment.

6) IMFR est une zone contenant une information relative à la fréquence d'exécution du segment, c'est-à-dire une estimation du nombre de fois que le segment sera exécuté lors d'une exploitation du programme.

Exemple : reprenons la sous-routine TALLEY et montrons la table TDS pour cette routine.



7; et ainsi de suite.

* Ces notions sont définies au chapitre III.

Déclaration en Algol-Jonquille-74 :

[1 : NVS] STRUCTURE TDS (INTEGER N, REFERENCE POINTERTSQ, REFERENCE PTLS, REFERENCE PTLP, PTPR, INTEGER IMFR);

Remarque

Dans l'exemple précédent, IMFR représentait une information de fréquence recueillie dans le programme source lui-même. Ceci n'étant pas toujours possible, des méthodes de mesures ont été développées. Ces méthodes se basent sur une observation dynamique du programme en période de test (cf. /KNUTH 71/). Il existe des modèles théoriques faisant appel aux "Processus stochastiques" et qui sont actuellement développés dans divers centres de recherches. Nous citons ici à titre d'information un ensemble d'articles et de "Ph.D.Thesis" traitant ces sujets.

- Cerf, V.G. "Measurement of Recursive Programs", Ph.D. Thesis, School of Engineering and Applied Sciences, University of California, Los Angeles, Report 70-43, May 1970.

- Ingalls, D., "FETE - A FORTRAN Execution Time Estimator".

- Johnston, T.Y. and Johnson, R.H., "Program performance measurement", SLAC User Note 33, revision 1, April 1970, Standford, California.

- Russell, E.C., Jr, "Automatic Program Analysis", Ph.D. Thesis, School of Engineering and Applied Science, University of California, L.A., California, Report 69-12, March 1969.

- Wichmann, B.A., "Some Statistics from ALGOL programs", National Physical Laboratory, Central Computer Unit, Report 11, August 1970.

-"Measurement based on Automatic Analysis of Fortran Programs", E.C. Russell and Estrin, Spring Joint Computer Conference 69.

- Wichmann, B.A., "A comparison of Algol 60 execution speeds", National Physical Laboratory, Central Computer Unit Report 3, January 1969.

Nous n'aborderons pas, dans cette étude, ce problème qui est sans doute un problème fondamental que doit se poser tout implémenteur d'un compilateur optimiseur.

1.2.4. Table de séquence TSQ

Cette table donne la séquence des instructions intermédiaires composant le programme source. Elle représente donc le programme source traduit en instructions intermédiaires.

La table est formée d'une séquence linéaire de pointeurs α_i ($i \in N^+$) vers la table des instructions intermédiaires (1.2.5). α_i est donc en quelque sorte une référence à l'instruction intermédiaire qui aurait dû être écrite à cet endroit.

Le début d'un segment dans cette table est délimitée par un pseudo-pointeur (par exemple $\uparrow\uparrow\uparrow$), c'est cette entrée dans TSQ qui est pointée par le γ_n correspondant.

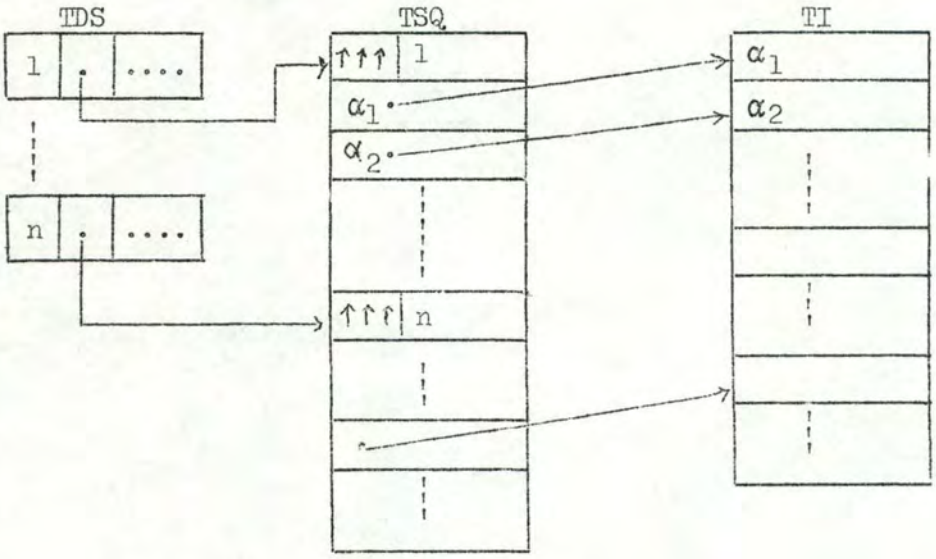
Format général d'une entrée dans TSQ :

1) α_i ; n — n est un entier dont la signification est décrite dans le chapitre II.

2) $\uparrow\uparrow\uparrow$; n — n est le numéro du segment.

Déclaration en Algol-Jonquille-74 : voir chapitre II.

Schéma de la disposition :



1.2.5. Table des instructions TI

La table TI contient un seul exemplaire de chaque triple composant le programme source traduit en texte intermédiaire. Cela signifie par exemple

que le triple α_i : +, ARG 1; ARG 2 figurant à plusieurs endroits différents dans le texte du programme, ne se trouve qu'une seule fois dans la table des instructions.

Exemple : Considérons l'instruction Algol 60 suivante :

a := a * b + (c * d + a * b) * c * d

TSQ	TI
α_1; n	α_1: *; c; d
α_2; m	α_2: *; a; b
α_3; max (n,m) + 1 = e	α_3: +; α_1; α_2
α_1; n	α_4: *; α_3; α_1
α_4; max (n,e) + 1 = k	α_5: +; α_2; α_4
α_2; m	α_6: :=; a; α_5
α_5; max (m,k) + 1 = j	
α_6; j + 1	

Nous donnerons dans ce qui suit une liste d'instructions intermédiaires pouvant figurer dans TI. Cette liste n'est pas exhaustive; nous y avons mis celles que nous utiliserons dans la suite de ce travail.

1.2.5.1. Considérations générales

Toutes les instructions intermédiaires ont le format général

α_i: OP; ARG 1; ARG 2

représentant l'opération ARG 1 OP ARG 2.

1) α_i est l'identificateur intermédiaire de l'instruction, en ce sens qu'il est un moyen de référencer le résultat de l'opération dans une instruction intermédiaire ultérieure.

α_i interprété au niveau de la TSQ apparaît comme un pointeur vers une entrée relative à α_i dans TI. Cette entrée décrit l'instruction intermédiaire devant figurer explicitement à cet endroit du programme.

2) ARG 1 et ARG 2 peuvent être, suivant les cas,

- des références à des variables simples (1.2.1.1)
- des références à des constantes (1.2.1.2)
- des références à d'autres instructions intermédiaires (1.2.1.3)

- des références à des éléments d'un tableau (1.2.1.4)
- des références à des éléments postiches
- des références à la TDS
- des références à des procédures standards ou utilisateurs.

1.2.5.2. Opérations de base

Format général : α_i: OP; ARG 1; ARG 2 avec OP $\in$ { +, -, *, **, /; AND, OR, NOT, IM, EQ, =, $\neq$, < , > , $\leq$, $\geq$, ..}

1) Il s'agit des opérations arithmétiques ((+) addition, (-) soustraction, (*) multiplication, (**) exponentiation, (/) division), logiques ((AND) et, (OR) ou, (NOT) négation, (IM) implication, (EQ) équivalence) de comparaison ((=) égal, ($\neq$) pas égal, (<) strictement inférieur, ($\leq$) inférieur ou égal, (>) strictement supérieur, ($\geq$) supérieur ou égal), ou toute autre opération multiplication de tableau que le langage source admet. Tout dépend des objectifs que l'on se fixe.

2) OP est codé de telle façon qu'à partir de OP, on puisse retrouver le type des opérandes. Par exemple, une addition d'entiers aura pour code 1. Cette codification de OP permettra de retrouver le type du résultat d'une opération intermédiaire et facilitera également la génération du code machine.

3) Le type du résultat des opérations logiques et de comparaisons est booléen, tandis que le type du résultat des opérations arithmétiques est déterminé par la codification de OP.

1.2.5.2.1. Opérations arithmétiques

α_i: +; ARG 1; ARG 2	α_i:= ARG 1 + ARG 2
α_i: -; ARG 1; ARG 2	α_i:= ARG 1 - ARG 2
α_i: *; ARG 1; ARG 2	α_i:= ARG 1 * ARG 2
α_i:**; ARG 1; ARG 2	α_i:= ARG 1 ** ARG 2
α_i: /; ARG 1; ARG 2	α_i:= ARG 1 / ARG 2
instructions intermédiaires	interprétation des instructions

Exemple : integer p, k

a := p * k * (k + 2.0) / (k + 1) ** 2

α_1 : CVIR; k;↑ (CVRI = convertir réel en entier)

α_2 : +α_1; 2.0 (CVIR = convertir entier en réel)

α_3 : +; k; 1

α_4 : ** ; α_3; 2

α_5 : CVIR; α_4;↑

α_6 : **; p; k

α_7 : CVIR; α_6; ↑

α_8 : **; α_7; α_2

α_9 : CVIR; α_4;↑

α_{10}: /; α_8, α_9

α_{11}: :=; a; α_{10}

1.2.5.2.2. Opérations de comparaison

α_i: $\leq$; ARG 1; ARG 2	α_i:= ARG 1 $\leq$ ARG 2
α_i: < ; ARG 1; ARG 2	α_i:= ARG 1 < ARG 2
α_i: $\geq$; ARG 1; ARG 2	α_i:= ARG 1 $\geq$ ARG 2
α_i: > ; ARG 1; ARG 2	α_i:= ARG 1 > ARG 2
α_i: = ; ARG 1; ARG 2	α_i:= ARG 1 = ARG 2
α_i: $\neq$; ARG 1; ARG 2	α_i:= ARG 1 $\neq$ ARG 2
instructions intermédiaires	interprétation des instructions

1.2.5.3. Opérations logiques

α_i: AND; ARG 1; ARG 2	α_i:= ARG 1.AND.ARG 2
α_i: OR; ARG 1; ARG 2	α_i:= ARG 1.OR.ARG 2
α_i: NOT; ARG 1;	α_i:= .NOT.ARG 1
α_i: IM; ARG 1; ARG 2	α_i:= ARG 1 $\supset$ ARG 2
α_i: EQ; ARG 1; ARG 2	α_i:= ARG 1 $\equiv$ ARG 2
instructions intermédiaires	interprétation des instructions

AND — ARG 1 / ARG 2	0	1
0	0	0
1	0	1

OR — ARG 1 / ARG 2	0	1
0	0	1
1	1	1

NOT ARG 1	0	1
NOT ARG 1	1	0

IM \ ARG 1 / ARG 2	0	1
0	0	0
1	1	1

EQ \ ARG 1 / ARG 2	0	1
0	1	0
1	0	1

Exemple : IF ((A.EQ.B.OR.A.NE.C).AND.(A.EQ.D)) GOTO 15 16: .....

α_1: =; A; B (TRL = transfert logique)
α_2: $\neq$; A; C
α_3: OR; α_1; α_2
α_4: =; A; D
α_5: AND; α_3; α_4
α_6: TRL; α_5; (15, 16)

1.2.5.3. Opérations de transfert de "contrôle"

Deux formats sont possibles: le format conditionnel et le format inconditionnel.

1.2.5.3.1. Transfert conditionnel

Format général α_i: OPTR; ARG 1; λ_k

1) α_i s'interprète de la même façon qu'en 1.2.5.1, sauf qu'ici, il n'est qu'un pointeur vers TI.

2) OPTR est l'opérateur de transfert du "contrôle".

3) λ_k est un pointeur vers l'entrée dans TDS pour le segment qui est terminé par cette opération de transfert. Rappelons ici le format d'une entrée dans la TDS:

λ_k: n; γ_n; PTLS; PTLP; PTPR; IMFR.

PTLS pointait vers la liste des successeurs du segment. Dans cette liste, les numéros n des segments successeurs immédiats sont placés dans un certain

ordre. Cet ordre est dépendant du type de transfert conditionnel. N'oublions pas qu'un segment ne peut être terminé que par une et une seule opération de transfert Nous exposerons dans l suite l'ordre des numéros de segments dans la liste LS (Liste des Successeurs).

4) ARG 1 peut être une variable simple ou indicée, une constante ou une instruction intermédiaire.

1.2.5.3.1.1. Transfert logique

Format général de l'opération : α_i : TRL; ARG 1; λ_k.

Le premier numéro dans la liste LS est celui du segment vers lequel on branchera si ARG 1 possède une valeur booléenne <u>true</u>, le deuxième numéro est celui vers lequel on branche dans le cas contraire.

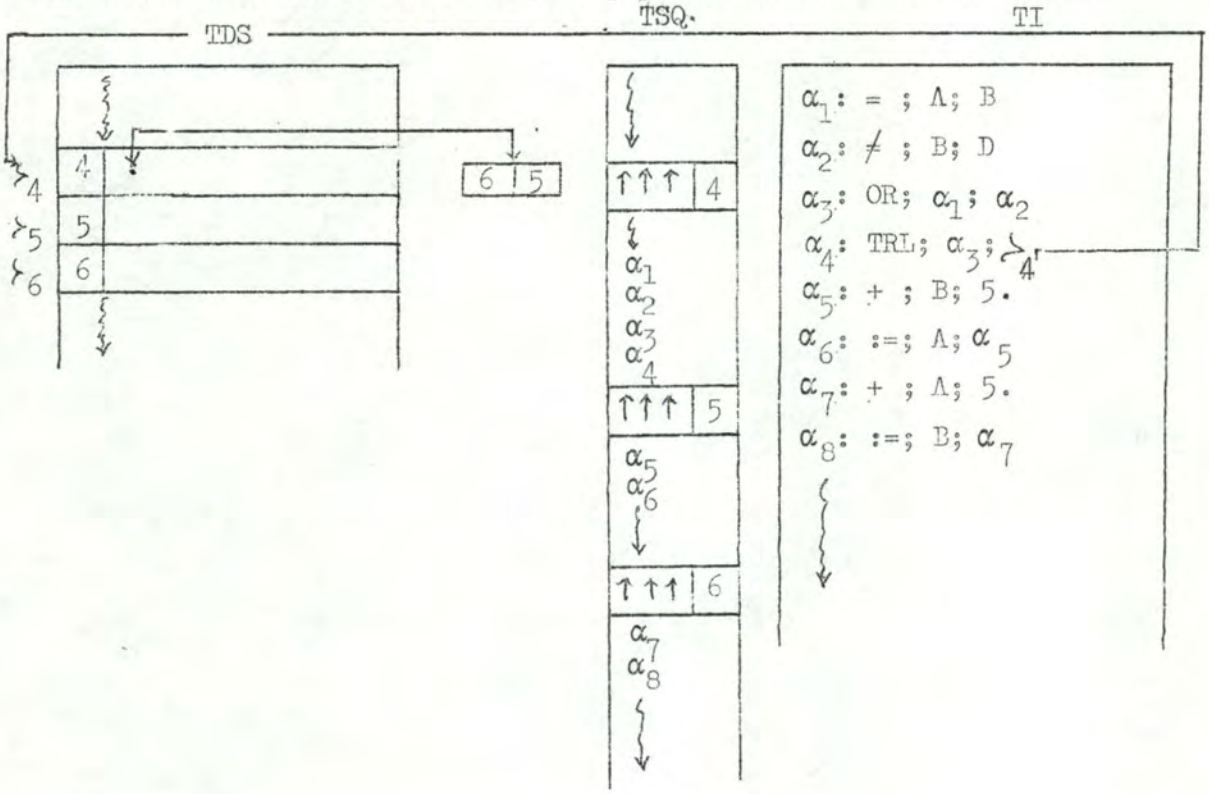
<u>Exemple</u> :

<u>IF</u> A = B ∨ B ≠ D <u>THEN</u> <u>GOTO</u> L20

L 10 : A : = B + 5.

L 20 : B : = A + 5.

sera traduit en forme intermédiaire pour donner le résultat suivant :



1.2.5.3.1.2. Transfert d'après valeur

Format général : α_i : TRV; ARG 1; $\succ_k$

ARG 1 représente la valeur entière n qui fera brancher vers le segment dont le numéro figure en $n^{ème}$ position dans la liste LS.

Exemple : En Algol, nous avons les "switches" et en Fortran les "goto" calculés. C'est pour ces instructions que TRV a été introduit.

goto acril [(I + 5) * 3]

GOTO (25, 30, 24, 76, 35), J

1.2.5.3.1.3. Transfert arithmétique

Format général α_i : TRA; ARG 1; $\succ_k$

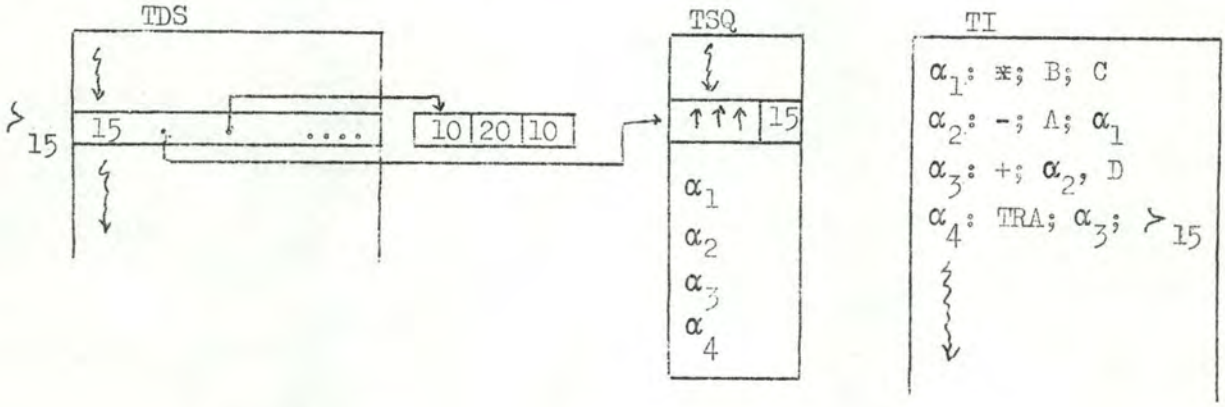
Arg 1 représente l'identificateur d'une quantité dont le signe fera brancher dans un sens ou l'autre.

Si ARG 1 < 0, alors on branche vers le segment dont le numéro figure en tête de la liste LS.

Si ARG 1 = 0, alors on branche vers le segment dont le numéro figure en deuxième position dans LS.

Si ARG 1 > 0, alors on branche vers le segment dont le numéro figure en troisième position dans LS.

Exemple : IF (A - B * C + D) 10, 20, 10 se trouvent dans le segment 15. 10 et 20 étant des numéros de segments.



1.2.5.3.2. Transfert inconditionnel

Format général : α_i : TRI; ↑↑ ; $\succ_k$

On branchera vers le segment dont le numéro unique se trouve dans la liste des successeurs du segment pointé par $\succ_k$ (dans TDS).

1.2.5.4. Opération d'assignation ou de définition d'une quantité source

Format général : α_i : := ARG 1; ARG 2.

1) ARG 1 peut être une variable simple ou indicée. Pour certains langages, ARG 1 pourrait être un tableau. Ce qui réduit la finesse du traitement.

2) ARG 2 peut être une variable simple ou indicée, une constante ou une instruction intermédiaire.

1.2.5.5. Opération complémentaire

L'arrêt du programme sera provoqué par une instruction du type α_i : STP; ↑↑ ; ↑↑

1.2.6. Les appels de procédures et les définitions de procédures

Dans ce paragraphe, nous envisagerons deux langages, "Algol 60" et "Fortran 4", et donnerons pour eux les formats des tables de définition des procédures et des tables des appels à ces procédures. Ensuite, nous envisagerons la séquence d'instructions intermédiaires générées lors de l'appel d'une procédure et lors de la définition d'une procédure. Nous ne présenterons pas ici une étude des différents types de "linkage" possibles. Celle-ci a été développée par A.P. Ershov (/ERSHOV 66/ et /ERSHOV 67/).

1.2.6.1. Table des appels de sous-routines, de fonctions TASF et appel de sous-routines et de fonctions

a) Table TASF

Dans le cas des procédures "Fortran 4", l'appel est un appel par référence, en ce sens que l'adresse du résultat du paramètre actuel (ou du paramètre actuel) est passée à la sous-routine.

Le format correspondant à une entrée dans TASF sera le suivant :

η_k : identificateur; ρ_j; NP; PTPAL; ADR; NPG; PTPGL; ADRR.

1) η_k s'interprète au niveau de la forme intermédiaire comme un pointeur vers l'entrée correspondante dans TASF pour l'appel de procédure.

2) identificateur est l'identificateur source de la procédure appelée.

3) ρ_j est l'identificateur intermédiaire de la procédure <u>value</u> (identificateur). Il s'interprète ici comme un pointeur vers la table de définition des procédures.

4) NP est le nombre de paramètres actuels de l'appel.

5) PTPAL est un pointeur vers la liste des paramètres (PAL) actuels de l'appel de procédure.

6) ADR est l'adresse à laquelle se trouve disponible le texte de la procédure, soit en langage source, soit en forme intermédiaire.

7) NPG est le nombre d'opérandes globaux de la procédure, c'est-à-dire le nombre de quantités figurant dans la liste des "COMMON" de la procédure <u>value</u> (identificateur).

8) PTPGL est un pointeur vers la liste des opérandes globaux PGL de la procédure <u>value</u> (identificateur).

9) ADRR est une zone contenant l'adresse de retour.

10) PAL : P 1 : T 1; P 2 : T 2; .....; P_{NP} : T_{NP}; ↑

Si le paramètre actuel est un tableau, une variable simple, une constante ou une procédure, alors P_i est la référence intermédiaire de la quantité correspondante.

Si le paramètre actuel est une expression arithmétique autre qu'une variable indicée, alors P_i est la référence à la case temporaire qui contient le résultat de l'expression.

Si le paramètre actuel est une variable indicée, alors P_i est la référence intermédiaire à la variable indicée, ceci nous permettra de réaliser une optimisation plus fine; en effet, si la partie indice possède une valeur connue, alors seul un élément univoquement déterminé est modifié par la procédure.

T_i est la référence intermédiaire à la case temporaire qui contient l'adresse de P_i. Ces cases doivent être contiguës en mémoire centrale. C'est la raison pour laquelle on écrira plutôt ce vecteur de cases temporaires sous la forme T [i].

11) PGL : PG_1; PG_2; ...; PG_{NPG}; ↑ ; est la liste des paramètres globaux

de la procédure value (identificateur). PG_i est la référence intermédiaire de l'opérande global.

b) Les instructions de la séquence d'appel

1° Les T [i] sont initialisés par des instructions intermédiaires du type α_i : ADR; T [i] : ARG 2.

- ADR est l'opérateur qui affecte à la case temporaire T [j] l'adresse de ARG 2 (adresse "run-time").

- ARG 2 peut être une variable simple (temporaire ou programmeur), une constante, une variable indicée ou un tableau.

- T [j] est un élément du vecteur temporaire T pour l'appel que l'on considère.

2° L'appel lui-même est indiqué par une instruction du format α_i : AP; η_k; ↑↑

- AP est l'opérateur d'appel qui, dans ce cas, s'interprète comme un branchement vers le code de la procédure.

- η_k est l'identificateur intermédiaire de l'appel, c'est-à-dire un pointeur vers la TASF.

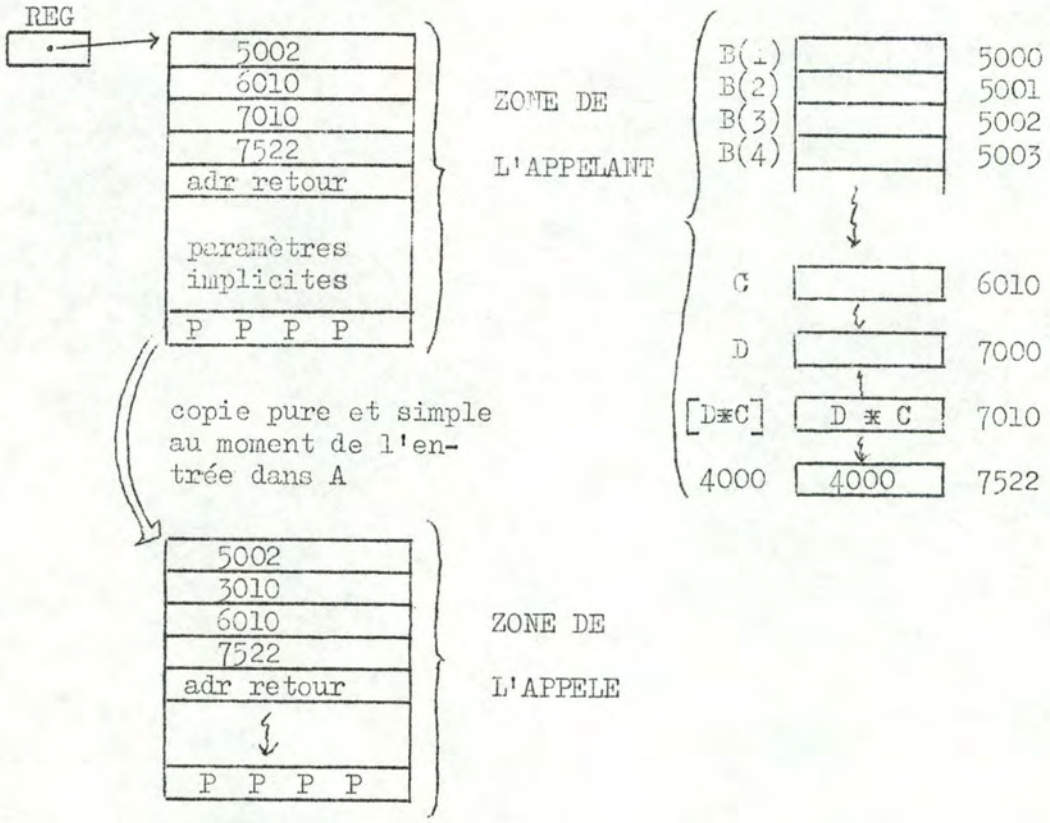
Exemple : soit la sous-routine "Fortran 4"

```
SUBROUTINE  A(X, Y, Z, W)
COMMON  E, F, G
```

et l'appel CALL A(B(3), C, D * C, 4000) dans le programme principal.

a) Organisation mémoire

(voir page suivante)



b) Texte intermédiaire

α_1: * ; D; C

α_2: :=; T; 1

α_3: ADR; T [1]; B [3]

α_4: ADR; T [2]; C

α_5: ADR; T [3]; T

α_6: ADR; T [4]; 4000

α_7: AP ; η_A; ↑↑

η_A : A; ρ_A; 4; $PTPAL_A$; DISC = 4 CYL = 10 TRACK = 16; 3; $PTPGL_A$; 100.

$PTPAL_A$: B [3] : T [1]; C : T [2]; T : T [3]; 4000 : T [4]; ↑

$PTPGL_A$: E; F; G; ↑

Remarques - Il y a, par appel, une entrée dans TASF.

- Les instructions ADR; T [i]; ARG 2 sont insérées dans le programme pour permettre une optimisation des séquences d'appels.

Exemple : soit le programme "Fortran 4" DO 1 I = 1,100

1 CALL SUB (A [I], B [I], K)

Seules les adresses de A [I] et de B [I] varient. Par conséquent, l'instruction α_3 : ADR; T [3]; K pourra être déplacée en dehors de la boucle, l'adresse de K ne variant pas et T [3] réfère toujours la même case temporaire.

c) Cas des fonctions

Pour le cas des fonctions, seule l'instruction d'appel diffère. Elle aura le format α_i : APF ; η_k; $\uparrow\uparrow$.

- APF est l'opérateur d'appel de fonctions. Il s'interprète comme un branchement vers la fonction avec retour du résultat dans α_i.

- η_k est le pointeur vers l'entrée dans TASF.

Exemple de traduction d'un programme FORTRAN 4

```
      DIMENSION X(10,10), Y(10,10), Z(10,10)
      CALL LECTURE (X,Y)
      DO 3 I = 1,10
      DO 3 J = 1,10
      Z(I,J) = 0.
      DO 3 K = 1,10
    3 Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
      CALL ECRITURE (Z)
      STOP
      END
```

1) TS

γ_1: I; I; $\overline{CO}$; $\overline{P}$; 000; .....;

γ_2: J; I; $\overline{CO}$; $\overline{P}$; 000; .....;

γ_3: K; I; $\overline{CO}$; $\overline{P}$; 000; .....;

2) TT

γ_4: X; R; $\overline{CO}$; P; REFERENCE; 2; 1 : 10, 1 : 10; .....;$\uparrow$

γ_5: Y; R; $\overline{CO}$; P; REFERENCE; 2; 1 : 10, 1 : 10; .....;$\uparrow$

γ_6: Z; R; $\overline{CO}$; P; REFERENCE; 2; 1 : 10: 1 : 10: .....;$\uparrow$

3) TC

γ_7: I; 1;

γ_8: I; 10;

γ_9: R; 0.;

4) TDS

$\succ_1$: 1; γ_1; PTLS 1; 0; 0; 1;

$\succ_2$: 2; γ_2; PTLS 2; PTLP 2; $\succ_1$; 10;

$\succ_3$: 3; γ_3; PTLS 3; PTLP 3; $\succ_2$; 100;

$\succ_4$: 4; γ_4; PTLS 4; PTLP 4; $\succ_3$; 1000;

$\succ_5$: 5; γ_5; PTLS 5; PTLP 5; $\succ_4$; 100;

$\succ_6$: 6; γ_6; PTLS 6; PTLP 6; $\succ_5$; 10;

$\succ_7$: 7; γ_7; 0 ; PTLP 7; $\succ_6$; 1;

PTLS 1 : 2
PTLS 2 : 3
PTLS 3 : 4
PTLS 4 : 4, 4, 5
PTLS 5 : 3, 3, 6
PTLS 6 : 2, 2, 7
PTLP 2 : 1, 6
PTLP 3 : 2, 5
PTLP 4 : 3, 4
PTLP 5 : 4
PTLP 6 : 5
PTLP 7 : 6

5) TC

α_1: ADR; T [1]; γ_4

α_2: ADR; T [2]; γ_5

α_3: AP ; η_1 ; ↑↑

α_4: := ; γ_1 ; γ_7

α_5: TRI; ↑↑ ; $\succ_1$

α_6: := ; γ_2 ; γ_7

α_7: TRI; ↑↑ ; $\succ_2$

α_8: * ; γ_2 ; γ_8

α_9: + ; α_8 ; γ_1

α_{10}: :=; γ_6 [α_9]; γ_9

α_{11}: :=; γ_3 ; γ_7

α_{12}: TRI; ↑↑ ; $\succ_3$

α_{13}: * ; γ_3; γ_8

6) TSQ

γ_1 : ↑↑↑ ; 1
: α_1 ; -
: α_2 ; -
: α_3 ; -
: α_4 ; -
: α_5 ; -

γ_2 : ↑↑↑ ; 2
: α_6 ; -
: α_7 ; -

γ_3 : ↑↑↑ ; 3
: α_8 ; -
: α_9 ; -
: α_{10} ; -

α_{14}:	+	; α_{13}; γ_1		: α_{11}	; -
α_{15}:	+	; α_8; γ_3		: α_{12}	; -
α_{16}:	*	; $\gamma_4[\alpha_{14}]$; $\gamma_5[\alpha_{15}]$	γ_4	: ↑↑↑	; 4
α_{17}:	+	; $\gamma_6[\alpha_9]$; α_{16}		: α_{13}	; -
α_{18}:	:=	; $\gamma_6[\alpha_9]$; α_{17}		: α_{14}	; -
α_{19}:	+	; γ_3 ; γ_7		: α_8	; -
α_{20}:	:=	; γ_3 ; α_{19}		: α_{15}	; -
α_{21}:	TRA	; α_{22}; $\succ_4$		: α_{16}	; -
α_{22}:	-	; γ_3 ; γ_8		: α_8	; -
α_{23}:	+	; γ_2 ; γ_7		: α_9	; -
α_{24}:	:=	; γ_2 ; α_{23}		: α_{17}	; -
α_{25}:	-	; γ_2 ; γ_8		: α_8	; -
α_{26}:	TRA	; α_{25}; $\succ_5$		: α_9	; -
α_{27}:	+	; γ_1 ; γ_7		: α_{18}	; -
α_{28}:	:=	; γ_1 ; α_{27}		: α_{19}	; -
α_{29}:	-	; γ_1 ; γ_8		: α_{20}	; -
α_{30}:	TRA	; α_{29}; $\succ_6$		: α_{22}	; -
α_{31}:	ADR	; TT [1]; γ_6		: α_{21}	; -
α_{32}:	AP	; η_2 ; ↑↑	$\succ_5$	: ↑↑↑	; 5
α_{33}:	STP	; ↑↑ ; ↑↑		: α_{23}	; -
				: α_{24}	; -
				: α_{25}	; -
				: α_{26}	; -
			$\succ_6$	: ↑↑↑	; 6
				: α_{27}	; -
				: α_{28}	; -
				: α_{29}	; -
				: α_{30}	; -
			$\succ_7$	: ↑↑↑	; 7
				: α_{31}	; -
				: α_{32}	; -
				: α_{33}	; -

7) TASF

$_1$:LECTURE; ρ_1; 2; PTPAL 1; ADR 1; 0; 0; ADRR 1

$_2$:ECRITURE; ρ_2; 1; PTPAL 2; ADR 2; 0; 0; ADRR 2

PTPAL 1 : γ_4 : T [1] ; γ_5 : T [2] ; ↑

PTPAL 2 : γ_6 : TT [1] ; ↑

1.2.7. Quelques avantages de la forme intermédiaire

1) Les triples indirects permettent de simplifier très fortement la recherche d'instructions formellement identiques.

2) Dans une instruction intermédiaire appartenant à un segment de numéro j, un α_i référencé est toujours défini dans le même segment de numéro j et avant la référence à α_i.

3) Il est tout aussi évident que si un α_i représente un résultat intermédiaire, ce α_i sera toujours référencé dans la partie des opérandes d'une opération intermédiaire et ceci dans le même segment que α_i.

4) Les triples indirects permettent plus de souplesse dans les procédures d'optimisation supprimant des instructions du code intermédiaire.

1.2.8. Remarque

La proposition de forme intermédiaire ci-dessus n'est valable que pour le langage Fortran 4. Pour Algol 60 et PL/1, il faudra compléter les tables TASF et l'ensemble des instructions intermédiaires. Notre but n'étant pas la conception d'un compilateur optimiseur Algol 60 ou PL/1 - car ce travail demanderait bien plus d'un an d'étude (la littérature à ce sujet est fort peu abondante) -, nous avons préféré ne pas nous y aventurer. Cependant, nous trouverons dans ce travail quelques idées utilisables dans le cas d'Algol 60. Nous signalons au lecteur qu'un compilateur optimiseur a été développé en URSS pour un langage du type ALGOL 60 par A.P. ERSHOV.

1.3. Dictionnaires des traductions et remarques sur la construction des concepts de base

1.3.1. Dictionnaire pour quelques instructions Fortran IV

Notations : S_i est le segment en cours d'élaboration
S_n est le segment correspondant à l'étiquette source n
S_{i+1} est le segment suivant S_i dans TSQ
$\succ_i$ est l'entrée dans TDS correspondant à S_i

GOTO n : l'instruction intermédiaire générée est TRI; $\uparrow\uparrow$; $\prec_i$, elle termine S_i et l'arc (S_i, S_n) est créé.

GOTO $(n_1, n_2, \ldots, n_k)$, J : TRV; J, $\succ_i$ termine S_i et les arcs (S_i, S_{n1}), (S_i, S_{n2}), ..., (S_i, S_{nk}) sont créés.

Idem pour GOTO J, $(n_1, n_2, \ldots, n_k)$

IF (e) n_1, n_2, n_3 : TRA; e; $\succ_i$ termine S_i et les arcs (S_i, S_{n_1}), (S_i, S_{n_2}) et (S_i, S_{n_3}) sont créés.

IF (e) statement :

(a) si "statement" n'est pas un GOTO

alors - un nouveau segment S_e est créé avec le code intermédiaire généré pour "statement"
- TRL; e; $\succ_i$ termine le segment S_i
- les arcs (S_i, S_{i+1}), (S_i, S_e) sont créés. (S_{i+1} est le segment qui débute avec le code pour l'instruction suivant le IF)

(b) si "statement" est un GOTO n;

alors - TRC; e; $\succ_i$ termine le segment S_i
- les arcs (S_i, S_{i+1}), (S_i, S_n) sont créés (S_{i+1} cf. (a))

DO n I = M 1, M 2, M 3 : - α_j: :=; I ; M 1
α_{j+1}: TRI; $\uparrow\uparrow$; $\succ_i$
termine le segment S_i
- les arcs (S_i, S_{i+1}) et (S_n, S_i) sont créés

- remarquons qu'il faut mémoriser M 2 et M 3 afin de pouvoir générer le code correspondant à I := I + M 2 et test (I - M 3)

Instructions arithmétiques : celles-ci complètent le segment en cours d'élaboration.

READ (a, b, END = n_1, ERR = n_2) liste :
- est traduit en un appel de la procédure Read
- les arcs (S_i, S_{n_1}), (S_i, S_{n_2}) sont créés.

Nous avons donné la correspondance programme source - forme intermédiaire pour la plupart des instructions Fortran IV - les autres instructions se traduisent d'une façon similaire.

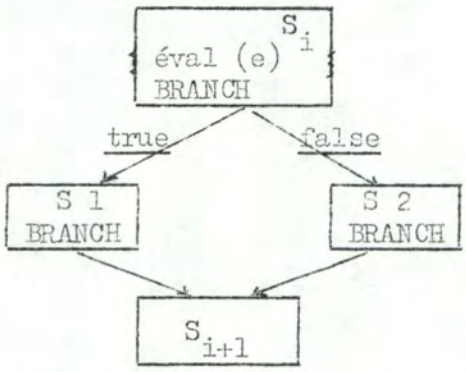
1.3.2. Dictionnaire pour quelques instructions Algol 60

goto e Si e est une étiquette définie dans le bloc où figure goto e, alors cfr. 1.3.1.

goto switch [j] : se traite de façon semblable à GOTO (n_1, n_2, ..., n_k), J si toutes les étiquettes figurant dans le switch sont définies dans le bloc où figure goto switch [j]

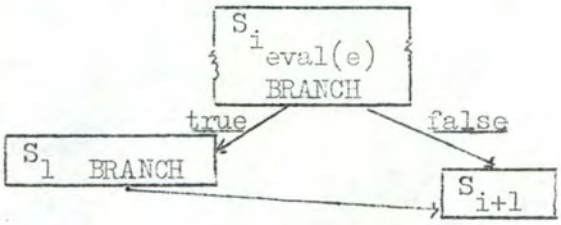
if e then S 1 else S 2 : nous donnons ici le graphe associé à cette instruction.

S_1 et S_2 n'étant pas des goto



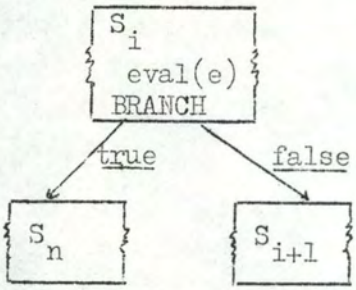
if e then S_1 : nous donnons également le graphe associé à cette instruction.

S_1 n'étant pas un goto

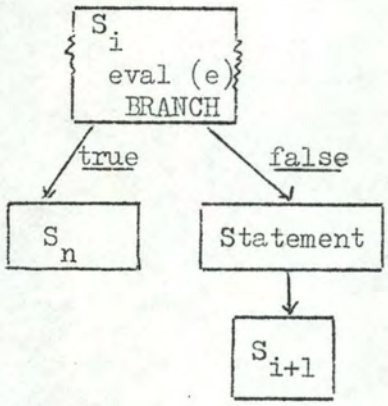


<u>for</u> i : = A_1 <u>step</u> A_2 <u>until</u> A_3 <u>do</u> : nous renvoyons le lecteur au chapitre 3 où sera discutée la traduction de cette instruction.

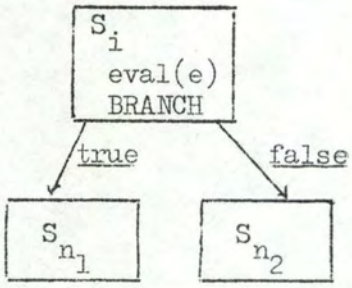
<u>if</u> e <u>then</u> <u>goto</u> n :



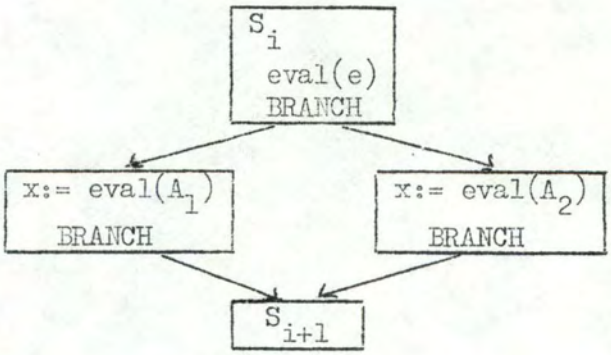
<u>if</u> e <u>then</u> <u>goto</u> n <u>else</u> statement :



<u>if</u> e <u>then</u> <u>goto</u> n_1 <u>else</u> <u>goto</u> n_2 :



x := <u>if</u> e <u>then</u> A_1 <u>else</u> A_2 :



Ceci est, nous semble-t-il, la façon la plus économique de réaliser la traduction de l'instruction.

Les instructions arithmétiques : complètent le segment en cours d'élaboration.

Nous venons de décrire quelques exemples de traduction d'instructions sources Algol 60. Pour les autres instructions, nous renvoyons le lecteur à un ouvrage traitant des techniques de compilation et plus particulièrement aux chapitres de ces ouvrages traitant des routines sémantiques.

1.3.3. Remarques sur la construction des concepts de base

Tout au long de ce travail, des remarques relatives aux routines sémantiques sont faites. Nous donnons ici quelques-unes de ces remarques. Nous avons préféré les insérer dans le texte aux endroits adéquats, afin de mettre en évidence l'importance et l'utilité de l'apport de ces routines à l'optimiseur.

1. Utilisation de la commutativité de certains opérateurs.
2. Utilisation des lois de Morgan pour la traduction des expressions booléennes.
3. Ordonnancement des opérandes d'une instruction source suivant un certain ordre.
4. Dans le cas suivant

   a := b

   remplacement de toute référence à a par une référence à b tant qu'une instruction modifiant la valeur de a ou de b n'est pas rencontrée.

etc...

Chapitre II

OPTIMISATION ORIENTEE SEGMENT D'INSTRUCTIONS

Le chapitre aura pour objectif de développer des optimisations locales, c'est-à-dire restreintes à un segment d'instructions où les problèmes sont beaucoup plus simples. Nous donnerons cependant toujours une définition générale de la technique d'optimisation envisagée, mais nous montrerons pourquoi il est difficile et parfois inutile de la réaliser d'une façon générale.

Plan

—— § ——

2.1. Propagation de constantes

Nous étudierons dans ce paragraphe la propagation de constantes. Nous en donnerons une définition générale, puis au moyen d'exemples, nous introduirons l'algorithme qui est décrit dans notre langage Algol-Jonquille-74. Nous effectuerons ensuite une critique de l'algorithme et dégagerons quelques améliorations que l'on peut y apporter.

Notre apport personnel a consisté en le développement de la technique de "propagation de constantes" pour des variables indicées. Etant donné le temps restreint qui nous a été donné, il nous a été impossible de vérifier l'exactitude de notre algorithme.

Mis à part le traitement des variables indicées, l'ensemble de ce paragraphe est basé sur les publications de Gries et de Allen (1).

2.1.1. Définition

Afin de définir la "propagation de constantes" d'une façon constructive, nous définirons tout d'abord deux processus P_1 et P_2.

2.1.1.1. Définition de P_1

P_1 est le processus

- qui, pour des opérations intermédiaires (ØP: ARG 1; ARG 2), exécute - à la compilation du programme source - les opérations intermédiaires dont les valeurs des deux opérandes sont connues; tel que, quel que soit un chemin allant de l'entrée vers l'opération, suivi lors de l'exploitation du programme, l'exécution de l'opération délivre la même valeur que celle obtenue par le processus P_1 pour l'opération intermédiaire; - qui, pour des assignations intermédiaires (:=; ARG 1; ARG 2), si ARG 2 a une valeur connue, mémorise temporairement à la compilation l'assignation à ARG 1 d'une valeur connue; si ARG 2 n'a pas de valeur connue, mémorise temporairement que la valeur actuelle de ARG 1 est quelconque. La valeur connue de ARG 2 doit être telle que cette valeur soit aussi la valeur de ARG 2 à l'exploitation du

---

(1) F.E. ALLEN, PRØGRAM ØPTIMIZATIØN.
David GRIES, CØMPILER CØNSTRUCTIØN FØR DIGITAL CØMPUTERS.

programme, quel que soit le chemin suivi dans le programme depuis le point d'entrée jusqu'à l'assignation en question.

2.1.1.2. Définition de P_2

P_2 est le processus

- qui remplace par sa valeur toute référence à une opération intermédiaire, ou à une variable simple, ou à un élément d'un tableau ayant une valeur calculée ou mémorisée par le processus P_1;
- qui supprime du programme source en forme intermédiaire toute occurrence d'une opération intermédiaire ayant une valeur connue, calculée par le processus P_1.

2.1.1.3. Définition de la "propagation de constantes"

Nous définirons la"propagation de constantes" comme étant le processus P_3 qui est la combinaison des processus P_1 et P_2.

2.1.2. Critique de la définition

Cette définition non formalisée a l'avantage de mettre en évidence les deux processus dominants dans la propagation de constantes. Elle est aussi inspirée par le souci de garantir l'équivalence sémantique entre le programme non optimisé et le programme obtenu en appliquant le processus P_3 au programme non optimisé.

De plus, par son énoncé, elle exige que l'optimisation du programme, par application du processus P_3, se déroule sur une machine équivalente à celle sur laquelle se déroule l'exploitation du programme.

Nous apercevons tout de suite la difficulté de l'application d'une telle définition dans toute sa généralité. Ceci est dû essentiellement d'une part à la difficulté d'énumérer tous les chemins dont parle la définition et d'autre part à la structure des langages évolués.

Soit l'extrait de programme Algol 60 suivant :

```
Begin   real  b;
        Begin  real x; switch acril [-----]
               procedure p (b,a); switch a; real b
                     Begin  b := 1.25;
                            goto a [i]
                     end
                     begin  real  y, z;
                         x : = 3. ;
                         y : = 5. ;
                         p (b, acril);
                     LO : z : = x + y;
                     end;
        end;
end.
```

Il est difficile de déterminer de façon statique (à la compilation) si, au point LO du programme, les variables x et y ont été modifiées ou non, ceci étant dû essentiellement à la structure du langage ALGOL 60, et plus particulièrement à la définition ALGOL 60 des procédures. Ceci est d'autant plus vrai s'il existe,dans la procédure p(b,a), des appels à d'autres procédures.

2.1.3. Restriction de l'application de P_3

C'est pour les raisons énoncées ci-dessus que nous n'allons envisager l'application du processus P_3 qu'aux segments d'instructions.

Dans un segment d'instructions, les seuls chemins possibles sont linéaires et nous pouvons facilement connaître de façon univoque l'évolution de l'état d'une variable source sur ces chemins.
En effet, il est toujours possible de dire en un point du segment (même en supposant non défini, à l'entrée du segment, l'état de toute variable du

programme) si la variable possède, à la compilation, une valeur connue univoquement.

2.1.4. Exemples

Nous allons considérer maintenant quelques exemples à partir desquels nous tâcherons de dégager la structure de l'algorithme. Nous utiliserons dans ces exemples une forme intermédiaire constituée de triples.

2.1.4.1.

J : = 2 + 2;
J : = 5;
A : = 8.2 + ;
C : = SIN (3.14159265/2.);
B : = TRUE;
D : = TRUE ∧ B;

Segment d'instructions source S
(a)

α_1 (: = ; J ; 4)
α_2 (: = ; J ; 5)
α_3 (: = ; A ; 13.2)
α_4 (: = ; C ; "1.")
α_5 (: = ; D ; TRUE)

P_3 (S) (c)

α_1 (+ ; 2 ; 2)
α_2 (: = ; J ; α_1)
α_3 (: = ; J ; 5)
α_4 (APS ; FLOAT ; J)
α_5 (+ ; 8.2 ; α_4)
α_6 (: = ; A ; α_5)
α_7 (/ ; 3.14159265 ; 2.)
α_8 (APS ; SIN ; α_7)
α_9 (: = ; c ; α_8)
α_{10}(: = ; B ; TRUE)
α_{11}(∧ ; TRUE ; B)
α_{12}(: = ; D ; α_{11})

Forme intermédiaire de S
(b)

A partir de cet exemple, nous pouvons faire les remarques suivantes.

1° La "propagation de constantes" s'applique essentiellement aux opérateurs arithmétiques et logiques.

2. Elle s'applique également aux opérateurs de conversion (FLOAT de l'exemple Réel ⟷ Entier).

3° Il est souhaitable qu'elle s'applique également aux appels de fonctions standards(sinus, cosinus, etc...) lorsque la valeur de l'argument de cette fonction est connue. Ceci implique que durant la phase d'optimisation, l'optimiseur dispose de l'adresse du code objet de la fonction standard.

4° (c) est le code interne optimisé; on remarque que des instructions intermédiaires ont disparu et donc que nous devons nous donner des moyens pour faire disparaître ces instructions intermédiaires. On remarque aussi que des instructions intermédiaires nouvelles doivent être introduites.

5° Des constantes disparaissent en ce sens qu'elles ne sont plus référencées dans le segment d'instructions, tandis que d'autres sont nouvelles. Nous devons donc nous donner un système de mise à jour de la table des constantes si nous désirons aussi minimiser le nombre de constantes réservées en mémoire principale.

2.1.4.2.

A : = 6 ;	α_1 (: = ; A ; 6)	α_1 (: = ; A ; 6)
B : = 7 ;	α_2 (: = ; B ; 7)	α_2 (: = ; B ; 7)
C : = A * B ;	α_3 (* ; A ; B)	α_3 (: = ; C ; 42)
D : = C / 2 ;	α_4 (: = ; C ; α_3)	α_4 (: = ; D ; 21)
	α_5 (/ ; C ; 2)	
	α_6 (: = ; D ; α_5)	
Segment d'instructions sources S (a)	Forme intermédiaire de S (b)	P_3(S) (c)

Nous remarquons que, dans le programme oprimisé, les opérations α_3 de (b) et α_5 de (b) ont disparu. Implicitement, nous avons supposé que nous nous étions donné un moyen de mémoriser la valeur actuellement connue d'une variable. De cette façon, si nous pouvons trouver de façon simple et rapide que A = 6 et que B = 7, il est possible de déterminer que α_3 de (b) vaut 42. De même, le fait que l'opération intermédiaire α_4 de (b) soit devenue (dans P_3(S)) α_3 de (c) suppose que nous possédons un moyen simple et efficace permettant de retrouver la valeur actuellement connue pour un triple (α_3 de (b) ici).

Ces considérations nous amèneront à introduire deux tables dont la description est donnée au paragraphe 2.1.5.

2.1.4.3.

I : = 5 ;	α_1(: =; I; 5)	α_1(: =; I; 5)
J : = 4	α_2(: =; J; 4)	α_2(: =; J; 4)
A [I] : = 3 ;	α_3(: =; A [I] ; 3)	α_3(: =; A [I]; 3)
B [J] : = 4 ;	α_4(: =; B [J] ; 4)	α_4(: =; B [J]; 4)
C [K] : = A [I] * B [J] ;	α_5(* ; A [I] ; B [J])	α_5(: =; C [K]; 12)
	α_6(: =; C [K] ; α_5)	
Segment d'instructions sources S	Forme intermédiaire de S	P_3(S)
(a)	(b)	(c)

Nous remarquons que l'opération intermédiaire α_5 de (b) a disparu de P_3(S) et que α_6 de (b) a été transformée en α_5 de (c). Ceci implique que nous nous sommes donné le moyen de mémoriser qu'un élément déterminé d'un tableau possédait une valeur connue, identique à celle qu'il contiendrait à cet endroit du programme lors de l'exploitation de celui-ci.
Ceci nous amènera à introduire une troisième table qui sera décrite dans le paragraphe 2.1.5.

2.1.5. Tables de l'algorithme (procédure FOLDING)

Pour réaliser les fonctions de mémorisation du processus P_1 et les fonctions remplacement du processus P_2, nous introduirons trois tables : STKVAL, STKIVAL, STKIVVAL.

2.1.5.1. STKVAL

STKVAL est le STACK des variables simples ayant une VALEUR actuellement connue. Le format d'une entrée dans STKVAL est le suivant :

(IDEN)	(VAL)

IDEN : contiendra la référence intermédiaire à la variable simple, c'est-à-dire le pointeur vers la table des variables simples où nous trouverons les renseignements sur cette variable.

VAL : contiendra la VALEUR actuellement connue pour la variable simple.

Dans notre langage ALGOL-JONQUILLE-74, la déclaration d'une telle table est la suivante :

[1 : NV] STRUCTURE STKVAL (REFERENCE IDEN, EXCEPT (REFERENCE, STRUCTURE)VAL).
Par la déclaration

REFERENCE PTSTKVAL FOR STKVAL; nous lui associons un pointeur du même genre que les variables index en COBOL, en ce sens que toute addition ou soustraction d'une unité doit être comprise comme étant une addition ou soustraction d'une unité égale à la longueur d'une entrée dans STKVAL.

Définition mathématique de STKVAL :

STKVAL est l'ensemble des couples (V,v) où V est un identificateur de variable simple et v la valeur actuellement connue de cette variable.

2.1.5.2. STKIVAL

STKIVAL est le STACK des opérations INTERMEDIAIRES ayant une VALEUR actuellement connue. Le format d'une entrée dans STKIVAL est le suivant :

(IDEN)	(VAL)
α_i	v

IDEN : contiendra une référence à une opération intermédiaire dont la valeur actuelle est connue.

VAL : contiendra cette valeur.

Déclaration en ALGOL-JONQUILLE-74 :

[1 : NI] STRUCTURE STKIVAL (REFERENCE IDEN, EXCEPT (REFERENCE, STRUCTURE) VAL).

Déclaration du pointeur associé dans notre langage :

REFERENCE PTSTKIVAL FOR STKIVAL.

Définition mathématique de STKIVAL :

STKIVAL est l'ensemble des couples (α_i, v_{α_i}) où α_i est un identificateur d'opération et v_{α_i} la valeur actuellement connue de α_i.

2.1.5.3. STKIVVAL

STKIVVAL est le "STACK" des VALEURS actuellement connues pour des VARIABLES INDICEES. Une entrée dans cette table a le format suivant :

(IDEN)	(SUBS)	(VAL)

IDEN : contiendra le pointeur vers la table des définitions de tableaux

SUBS : contiendra la valeur de la partie variable du calcul d'adresse d'un élément du tableau référencé par IDEN

VAL : contiendra la valeur de l'élément IDEN [SUBS] du tableau référencé par IDEN.

Il est important, à ce niveau, de remarquer que dans SUBS, il faut mettre la valeur de la partie variable du calcul d'adresse afin d'identifier de façon univoque un élément bien déterminé du tableau.
En effet, si nous considérions le format dans lequel SUBS contiendrait la référence au résultat du calcul de la partie variable, nous serions conduits à prendre A [J] comme étant le même élément du vecteur A que A [J] pour J ayant une valeur différente. De plus, A [5] := 20 modifie uniquement le 5me élément du vecteur A tandis que A [J] : = 20 peut, a priori, modifier n'importe quel élément du vecteur A.

Déclaration en ALGOL-JONQUILLE-74 :

[1 : NVI] <u>STRUCTURE</u> STKIVAL (<u>REFERENCE</u> IDEN, <u>INTEGER</u> SUBS, <u>EXCEPT</u> (<u>REFERENCE</u>, <u>STRUCTURE</u>) VAL);

Déclaration du pointeur associé :

<u>REFERENCE</u> PTSTKIVVAL <u>FOR</u> STKIVVAL.

<u>Définition mathématique de STKIVVAL</u>

STKIVVAL est l'ensemble des triples (T, i, v) où T est un identificateur de tableaux, i la valeur du calcul de la position d'un élément dans T si on considère celui-ci comme un vecteur et v la valeur de T [i].

2.1.5.4. <u>Remarques</u>

A) Dans notre algorithme, nous avons considéré que l'organisation de ces trois tables était séquentielle.
Ceci peut paraître très inefficace, mais il faut remarquer que ces trois tables n'existent que durant l'examen d'un segment d'instructions par le processus P_3 et que, comme nous l'avons déjà dit précédemment, la longueur d'un segment d'instructions est en moyenne de 3 instructions sources. Dès lors, ces tables ne seront en moyenne jamais très grandes et une organisation plus complexe ne se justifierait que très rarement.

B) Nous n'avons considéré jusqu'à présent que deux sortes de données: des variables simples et des tableaux. Néanmoins, les structures ne posent pas plus de problème que des variables simples, vu que tout élément d'une structure peut être adressé d'une façon statique. De plus, le fait que certains éléments d'une structure puissent être répétés ne pose pas plus de problème que les variables indicées.

C) A l'entrée dans la procédure Folding, les tables STKVAL, STKIVAL, STKIVVAL sont vides. La procédure Folding est appelée chaque fois que l'on examine un nouveau segment d'instructions.

2.1.6. Description des nécessités du processus P_3, restreint aux segments d'instruction, tel que nous l'avons programmé en ALGOL-JONQUILLE-74

2.1.6.1. Pour exprimer la fonction de suppression du processus P_2, nous compléterons le format d'une entrée dans TSQ en lui ajoutant un élément binaire: BIT.

α_i/↑↑↑	- / n°	
(IDEN)	(DEP-V-NOS)	(BIT)

Cet élément binaire BIT sera positionné à TRUE si l'opération intermédiaire doit donner lieu à une génération, en code machine, de son équivalent sémantique. Il sera positionné à FALSE dans le cas contraire.

Nouvelle déclaration de TSQ en ALGOL-JONQUILLE-74 :

[1 : LII] STRUCTURE TSQ (REFERENCE IDEN, INTEGER DEP-V-NOS, LOGICAL BIT).

Déclaration du pointeur associé :

REFERENCE NOW FOR TSQ.

Remarque : Avant toute optimisation, l'élément binaire BIT est toujours initialisé à TRUE.

2.1.6.2. Dans l'exemple 2.1.4.1, nous avons mis en évidence l'utilité d'un système de mise à jour de la table des constantes (TC). Pour réaliser cette mise à jour de TC, nous compléterons le format d'une entrée dans TC en ajoutant à cette entrée un élément binaire : DO.

DO sera positionné à TRUE lorsque, lors de la génération de code machine, la constante doit être générée.

DO sera positionné à FALSE dans le cas contraire.

Nouvelle déclaration de TC dans le langage ALGOL-JONQUILLE-74

[1 : NGG] STRUCTURE TC (CHARACTER (n) ATRIBUT) EXCEPT (REFERENCE, STRUCTURE) VAL, LOGICAL DO).

2.1.6.3. Les zones de travail

2.1.6.3.1. ITEM contient le texte de l'instruction intermédiaire en cours d'examen.

(OP)	(ARG 1)		(SUBS 1)		(ARG 2)		(SUBS 2)	
	CODE	POINTER	CODE	POINTER	CODE	POINTER	CODE	POINTER
					x			

Déclaration en ALGOL-JONQUILLE-74

STRUCTURE ITEM (INTEGER OP, STRUCTURE ARG 1 (INTEGER CODE, ---, REFERENCE POINTER)
, STRUCTURE SUBS 1 (idem)
, STRUCTURE ARG 2 (idem)
, STRUCTURE SUBS 2 (idem).

Rappel : L'élément marqué x, en ARGOL-JONQUILLE-74, est référencé de la façon suivante : CODE.ARG 2.ITEM.

2.1.6.3.2. OP est une zone entière qui contient le code de l'instruction intermédiaire en cours d'examen.

2.1.6.3.3. IDEM contient la référence de l'instruction intermédiaire en cours d'examen.

2.1.6.3.4. POINTER 1 : = POINTER.ARG 1.ITEM;
POINTER 2 : = POINTER.ARG 2.ITEM;

V_1 contient la valeur de ARG 1 (1er opérande)

V_2 contient la valeur de ARG 2 (2me opérande).

2.1.6.3.5. POINTER 1 : = POINTER.SUBS 1.ITEM;
POINTER 2 : = POINTER.SUBS 2.ITEM;

VI_1 contient la valeur de SUBS 1

VI_2 contient la valeur de SUBS 2.

2.1.6.3.6. Il existe encore d'autres zones de travail mais les principales ont été citées. Lorsqu'il examinera le détail de l'algorithme, le lecteur pourra facilement en déduire le sens et le rôle qu'elles y jouent.

2.1.7. Structure de la procédure FOLDING

(La procédure FOLDING est le processus P_3 restreint aux segments d'instructions)

2.1.7.1. Règles générales pour le traitement des variables indicées

Le traitement des variables indicées est sans doute plus complexe que celui des variables simples, mais lorsque la valeur de la partie variable du calcul d'adresse d'un élément d'un tableau est déterminée de façon exacte, une variable indicée se comporte de la même façon qu'une variable simple pour le processus.

Dès lors, pour que le processus P_3 puisse identifier de façon statique un élément déterminé d'un tableau, il faut qu'il connaisse de façon exacte la valeur des indices et a fortiori le nom du tableau.

Exemple

```
Integer array A [1 : 30]
        ↓
(1) L 1 : A [J] : = 20;
        ↓
(2) L 2 : A [J] : = 30;
        ↓
```

Pour que les assignations sources (1) et (2) réfèrent le même élément du vecteur A, il faut et il suffit que la valeur de J soit identique en ces deux endroits L 1 et L 2 du programme.

Pour que deux références A [SUBS 1] et A [SUBS 2] s'adressent au même élément du tableau A, il faut et il suffit que les valeurs de SUBS 1 et de SUBS 2 soient identiques.

Ceci est le principe de base sur lequel s'appuie la procédure FOLDING en ce qui concerne le traitement des variables indicées.

En général, le traitement de toute opération intermédiaire impliquant des variables indicées (VAR [SUBS]) aura la forme logique suivante.

Si la valeur de SUBS est connue, alors suite du traitement
sinon abandon du traitement.

Une variable ARG [SUBS] possède une valeur connue si et seulement s'il existe dans STKIVVAL une et une seule entrée dont les deux premiers éléments contiennent respectivement ARG et la valeur actuellement connue pour SUBS.

2.1.7.1. Règle générale pour le traitement des références à des instructions intermédiaires

Une instruction intermédiaire possède une valeur connue si et seulement si elle est une opération constante ou figure une et une seule fois dans STKIVAL.

2.1.7.2. Règle générale pour le traitement des variables simples

Une variable simple possède une valeur connue si et seulement si elle figure une et une seule fois dans STKVAL.

2.1.7.3. Computations

Nous appelons "computation" toute instruction intermédiaire de la forme OP; ARG 1; ARG 2

où OP est un opérateur appartenant à l'ensemble des opérateurs arithmétiques ou logiques permis,

où ARG 1 est le premier opérande,

où ARG 2 est le deuxième opérande,

ARG 1 et ARG 2 pouvant être des références à une constante, à une variable simple, à un élément d'un tableau ou à une opération intermédiaire qui précède toujours l'instruction intermédiaire que l'on examine dans le segment d'instructions.

Forme logique du traitement :

Si ARG 1 et ARG 2 ont une valeur connue, alors

Début - marquer l'opération comme devant ne pas être générée;
- exécuter l'opération intermédiaire;
- si il existe dans STKIVAL une entrée correspondante à l'opération - alors modifier la partie VAL de cette entrée
- sinon créer dans STKIVAL une nouvelle entrée pour cette opération

Fin

sinon :

Début - remplacer dans l'opération en examen toute référence à une opération intermédiaire ayant une valeur connue par la référence à la valeur connue;

- si il y a eu remplacement

  alors générer une nouvelle entrée dans TI et modifier TSQ;

- enlever de STKIVAL l'entrée relative à l'opération intermédiaire en examen;

Fin

Prendre en considération l'instruction intermédiaire suivante.

2.1.7.4. Assignations

Nous appelons "assignation" une instruction intermédiaire de la forme

: =; ARG 1; ARG 2

où : = est l'opérateur d'assignation,

où ARG 1 est une variable simple et la "cible" de l'assignation,

où ARG 2 est, sa valeur, l'"objet" de l'assignation.

ARG 2 peut être une référence à une constante, à une variable simple, à une variable indicée ou à une opération intermédiaire.

Forme logique du traitement :

- Si ARG 2 a une valeur connue, alors

  Début - si ARG 1 figure dans STKVAL

  alors modifier la partie VAL de l'entrée correspondante

  sinon créer une nouvelle entrée dans STKVAL pour ARG 1 avec VAL = valeur de ARG 2

  Fin

  sinon supprimer de STKVAL l'entrée correspondante pour ARG 1 si elle existe;

- remplacer dans l'assignation en examen toute référence à une opération intermédiaire ayant une valeur connue par la référence à la valeur connue; s'il y a eu remplacement, alors générer une nouvelle entrée dans TI et modifier TSQ;

- prendre en considération l'instruction intermédiaire suivante.

2.1.7.5. Indexed assignations

Une "Indexed assignation" est une instruction intermédiaire du même type que "assignation", sauf que ARG 1 est une variable indicée. On a donc la forme : = ; ARG 1 [SUBS 1] ; ARG 2.

Forme logique du traitement

- Si SUBS 1 a une valeur connue, alors

```
Début - si ARG 2 a une valeur connue alors
        Début si ARG 1 [SUBS 1] figure dans STKIVVAL
              alors modifier la partie VAL de l'entrée corres-
                    pondante VAL = valeur de ARG 2
              sinon créer une nouvelle entrée dans STKIVVAL
                    pour ARG 1 [SUBS 1] avec VAL = valeur de
                    ARG 2;
        Fin
                                    sinon
              supprimer de STKIVVAL l'entrée correspondante
              pour ARG 1 [SUBS 1] si existe.
Fin

                        sinon
Début  supprimer de STKIVVAL l'entrée correspondante pour
       ARG 1 [SUBS 1] si elle existe plus toutes les entrées
       ayant IDEN = ARG 1
Fin;
```

- Remplacer dans l'opération intermédiaire en examen toute référence à une opération intermédiaire ayant une valeur connue par la référence à une valeur connue.
- Si il y a eu remplacement alors générer une nouvelle entrée dans TI et modifier TSQ;
- Prendre en considération l'instruction intermédiaire suivante.

2.1.7.6. APS (Appel de Procédure Standard)

Nous appelons un "APS" toute instruction intermédiaire du type APS; ARG 1; ARG 2;

où ARG 1 est la référence à la procédure standard,

où ARG 2 est l'argument de la procédure standard (peut être une variable indicée).

Forme logique du traitement :

Si ARG 2 a une valeur connue, alors

Début - exécuter ARG (ARG 2);
- si il existe dans STKIVAL une entrée correspondante à l'opération
  alors modifier la partie VAL de l'entrée
  sinon créer dans STKIVAL une nouvelle entrée pour cette opération;
- marquer l'opération comme ne devant pas être générée;

Fin

sinon

Début - remplacer dans l'opération en examen toute référence à une opération intermédiaire ayant une valeur connue par la référence à la valeur connue;
- si il y a eu remplacement alors générer une nouvelle entrée dans TI et modifier TSQ;
- enlever de STKIVAL l'entrée relative à l'opération intermédiaire en examen

Fin

Prendre en considération l'instruction intermédiaire suivante.

2.1.7.7. AP (Appel de Procédure)

Nous tenterons dans ce paragraphe de décrire le problème posé par les "Appels de Procédures". Dans l'ensemble de la littérature que nous avons consultée, ce problème est bien souvent passé sous silence, ou bien les solutions que l'on y apporte sont souvent très grossières.

Un appel de procédure n'est susceptible de modifier une variable d'un programme que si celle-ci figure dans la liste des paramètres actuels de l'appel ou si elle est un opérande global à la procédure. Nous écrivons "susceptible d'être modifiée" car il n'est pas sûr qu'une variable, qui est un opérande global ou qui figure dans la liste des paramètres actuels, sera nécessairement modifiée par l'appel de la procédure. Il faut dès lors envisager trois questions :

1° Comment agir sur les tables de la procédure FOLDING en fonction des différents types d'appel ?

2° Comment peut-on déterminer de façon simple et statique les opérandes globaux de la procédure appelée ?

3° Comment peut-on déterminer parmi l'ensemble des opérandes de la procédure (paramètres actuels plus opérandes globaux) ceux qui sont réellement modifiés par la procédure ?

Nous tâcherons de donner une réponse à la première question. Pour la deuxième, nous donnerons une ébauche de solution. Quant à répondre à la troisième .....

2.1.7.7.1. Problème des types d'appel

En fonction des différents types d'appel, nous présentons une ébauche de solution. Il s'agit bien d'une ébauche car les solutions présentées pourraient être raffinées si nous parvenions à résoudre la troisième question.

2.1.7.7.1.1. Appel par "référence"

Dans l'appel par référence, c'est l'adresse du paramètre actuel qui est passée à la procédure appelée. En fonction de la nature du paramètre actuel, on a les solutions suivantes :

a) Variable simple (VAR)

Solution grossière: supprimer de STKVAL l'entrée relative à la variable simple VAR.

Solution plus raffinée: si VAR est modifiable par la procédure
alors supprimer l'entrée relative à VAR dans STKVAL.
sinon laisser STKVAL tel qu'il est.

b) Variable indicée VAR [I]

Solution grossière: si I a une valeur connue
alors supprimer l'entrée correspondante à VAR [I] dans STKIVVAL
sinon supprimer les entrées dans STKIVVAL ayant IDEN = VAR.

Solution plus raffinée:

- si I a une valeur connue et si VAR [I] est modifiable par la procédure
  alors supprimer l'entrée relative à VAR [I] dans STKIVVAL;
- si I a une valeur connue et VAR [I] non modifiable par la procédure
  alors laisser STKIVVAL tel quel;

- si I n'a pas de valeur connue et VAR [I] modifiable par la procédure
  alors supprimer toutes les entrées dans STKIVVAL ayant IDEN = VAR.

c) Expression arithmétique (E.A.)

Ici le problème est identique à celui de la variable simple, puisque l'adresse passée à la procédure est celle d'une zone temporaire contenant la valeur de E.A.

2.1.7.7.1.2. Appel par "valeur" et par "dummy argument"

Comme il n'y a que la valeur du paramètre actuel qui est passée à la procédure, cet appel ne modifie en rien les tables STKVAL et STKIVVAL.

2.1.7.7.1.3. Appel par "résultat"

Cet appel retournant dans le paramètre actuel, le résultat d'un calcul effectué dans le corps de la procédure, nous supposerons ici que d'office après l'appel, le paramètre est modifié. Nous avons ainsi les solutions suivantes en fonction de la nature du paramètre.

a) Variable simple (VAR)

Supprimer l'entrée correspondant à VAR dans STKVAL.

b) Variable indicée (VAR [I])

Si I a une valeur connue
Alors supprimer l'entrée correspondant à VAR [I] dans STKIVVAL
Sinon supprimer les entrées dans STKIVVAL ayant IDEN = VAR.

2.1.7.7.1.4. Appel par "résultat-valeur"

Cet appel se traite de façon identique à l'appel par "résultat".

2.1.7.7.1.5. Appel par "nom"

Ce type d'appel étant de loin le plus complexe, nous n'apporterons ici que des solutions draconiennes. Il mérite à lui seul toute une étude particulière.

a) Variable simple (VAR)

Supprimer de STKVAL l'entrée correspondant à VAR.

b) Variable indicée de même que tableau

Supprimer de STKIVVAL toute entrée correspondant à l'identificateur du tableau.

c) Expression arithmétique E.A.

Supposons que l'implémentation se fait au moyen de "thunk" (Ingerman THUNKS, CACM, janvier 1961).
Dans le "thunk", l'expression arithmétique est évaluée et l'adresse de la valeur est passée à la procédure chaque fois que, dans le corps de la procédure, une référence est faite à l'E.A. L'évaluation de cette expression arithmétique est une séquence linéaire d'instructions intermédiaires dans notre modèle. Il en résulte que nous enlèverons de STKIVAL toutes les entrées relatives à des instructions intermédiaires intervenant dans le "thunk".
De même, toutes les entrées relatives à des variables simples ou indicées utilisées dans le "thunk" seront supprimées de STKVAL et STKIVVAL.

2.1.7.7.2. Opérandes globaux

Nous n'envisagerons ici que le cas de deux langages, à savoir ALGOL 60 et FORTRAN 4.

a) FORTRAN 4

Les seuls opérandes globaux d'une sous-routine FORTRAN 4 sont ceux qui figurent dans la liste d'un ordre COMMON. Si l'on peut répondre à la question 3, nous dirons qu'il faut supprimer de STKVAL et de STKIVVAL toutes les entrées qui correspondent à des éléments figurant dans la liste de l'ordre COMMON et qui sont modifiables par la sous-routine. Si l'on ne peut pas répondre à la question 3, nous dirons qu'il faut supprimer de STKVAL et de STKIVVAL toutes les entrées qui correspondent à tous les éléments figurant dans la liste de l'ordre COMMON.

b) ALGOL 60

La solution la plus simple est de vider les tables STKVAL, STKIVVAL et STKIVAL car a priori tout élément déclaré dans un bloc de niveau inférieur ou égal au niveau du bloc de la déclaration de la procédure (si nous supposons que l'élément n'a pas été redéclaré dans un bloc du niveau supérieur au niveau du bloc de sa déclaration), peut être un opérande global.

Exemple

```
niveau 1
[   real a
    2 real b
    [ 3 integer x ; procédure p (---) -----;
      [ 4 real c ;
      [     p(-----);
      -
    [
    2
    [
    [
[
```

a priori, a, b, x peuvent être des opérandes globaux de la procédure p(---).

De toute façon, dans notre modèle, une procédure ALGOL 60 terminera toujours un segment d'instructions. Ceci est sans doute très restrictif mais, vu que des "switchs" et des "labels" peuvent être des paramètres d'une procédure, l'adresse réelle de retour d'une procédure n'est pas nécessairement l'adresse de l'instruction suivant l'appel de la procédure, ce qui est le cas en FORTRAN 4.

Nous limiterons nos considérations à cette description du problème. Il est possible qu'une réponse à la troisième question soit présentée dans une annexe de travail où nous présenterons un algorithme d'analyse du flot des données dans un programme.

2.1.7.8. Branchements

Nous n'aborderons ici que les instructions intermédiaires de branchement que nous utilisons dans notre travail.

Une instruction de branchement est toujours la dernière instruction d'un segment d'instructions. Puisque nous supposons qu'à l'entrée d'un segment d'instructions, STKVAL, STKIVAL et STKIVVAL sont vides et que le seul chemin possible dans un segment d'instructions est celui qui va depuis l'entrée jusqu'à la sortie en passant séquentiellement par toutes les instructions du segment, nous pouvons écrire que :

1° si une variable figure dans STKVAL, avec une valeur v, cette valeur sera la valeur de la variable à la sortie du segment, quel que soit le chemin suivi dans le programme depuis l'entrée jusqu'au segment.

2° la conclusion 1° est valable pour des opérations intermédiaires;

3° la conclusion 1° est aussi vraie pour des variables indicées dont les indices ont des valeurs connues.

Dès lors, nous pouvons affirmer que les transformations qui suivent conservent l'équivalence sémantique.

1° TRR ; ARG 1 ; (Seg 1, Seg 2) transfert logique

Si ARG 1 a une valeur connue alors

Début si valeur (ARG 1) = TRUE

alors transformer le transfert en un transfert inconditionnel TRI ;↑↑ ; Seg 1

sinon transformer en TRI ;↑↑; Seg 2

Fin

sinon laisser le transfert tel quel.

2° TRV ; ARG 1 ; (Seg_1, ..., Seg_n) transfert d'après valeur

Si ARG 1 a une valeur connue alors

Début transformer le transfert en un transfert inconditionnel TRI ; ↑↑ ; Seg_{valeur} (ARG 1)

Fin

sinon laisser transfert tel quel.

3° TRA ; ARG 1 ; (Seg_1, Seg_2, Seg_3) transfert arithmétique

Si ARG 1 a une valeur connue alors

Début si Val (ARG 1) = 0, alors transformer en TRI ; ↑↑; Seg_1

si Val (ARG 1) < 0, alors transformer en TRI ; ↑↑; Seg_2

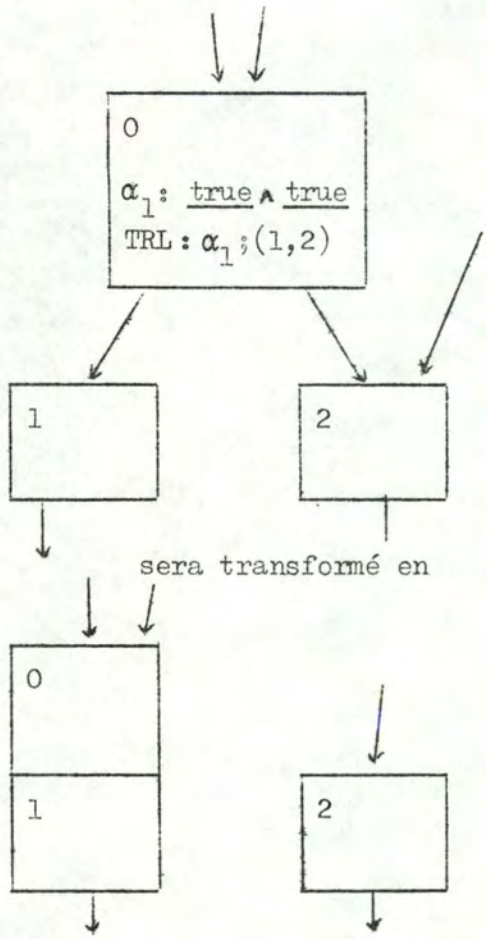
si Val (ARG 1) > 0, alors transformer en TRI ; ↑↑; Seg_3

Fin

sinon laisser le transfert tel quel

Il est à remarquer que ces transformations modifient et simplifient la structure du graphe associé au programme. Cette simplification pourrait nous amener à fusionner des segments d'instructions.

Exemple



Nous n'étudierons pas dans ce travail la réalisation d'un tel algorithme. Pourtant, nous pouvons nous apercevoir des avantages qu'il présente.

1° Il permettrait l'application de la procédure de Folding d'une façon plus globale. En effet, lorsque la procédure de Folding est amenée à examiner le segment 1, elle peut garder ses tables STKVAL, STKIVAL et STKIVVAL telles qu'elles se présentaient à la fin de l'examen du segment 0. Il n'y a pas que la procédure de Folding qui pourrait être appliquée d'une façon plus globale mais aussi l'élimination d'instructions redondantes.

2° La transformation de branchement conditionnel en branchement inconditionnel et la suppression de ce dernier permettent de diminuer le temps d'exécution d'un programme.

2.1.8. Discussion sur la "propagation de constantes"

On peut se poser la question de savoir si la propagation de constantes apporte une amélioration conséquente du programme.

Prenons comme support de notre discussion l'exemple suivant. Il n'est pas inhabituel pour un programmeur de définir au début de son programme une quantité PI : = 3.14159265 et ensuite d'utiliser, dans le reste de son programme, la quantité PI en lieu et place de 3.14159265 pour des raisons de facilité d'écriture et aussi de lisibilité du programme, par exemple

S : = PI * R ** 2 (1)

C : = 2 * PI * R (2)

En supposant que la quantité PI n'est plus redéfinie dans le programme, (2) peut trivialement se récrire

C : = 6.28318530 * R

qui n'est rien d'autre que la version optimisée de (2) par le processus P_3.

Nous avons gagné ainsi une multiplication de réels, ce qui veut dire que si (2) figurait dans le corps d'une boucle for ou DO, nous aurions gagné plus d'une multiplication de réels.

La procédure "Folding" présentée dans le paragraphe 2.1 n'envisage pas la propagation de constantes sous un angle aussi global mais les améliorations qu'elle apporte au programme peuvent être parfois fort appréciables.

Pour des opérations intermédiaires n'impliquant pas des variables indicées, l'analyse requise par la procédure "Folding" n'est ni énorme ni compliquée. Nous admettons cependant que pour des instructions intermédiaires impliquant des variables indicées, l'analyse est plus conséquente et plus complexe car il est nécessaire d'être très prudent.

Nous admettons également que l'optimisation présentée pour les branchements est assez aventureuse, en ce sens qu'elle modifie la structure du graphe. Si une telle modification devait être effectuée, cela pourrait provenir d'une erreur de logique du programmeur et il serait souhaitable de la lui signaler.

Il est important de s'assurer que le processus P_3 ne provoque pas d'erreurs à la compilation.

Exemple (extrait de Gries) IF FALSE THEN A : = 1/0

Un programme qui contiendrait cette instruction ne serait jamais interrompu à cet endroit pour une division par zéro, alors que la procédure "Folding" peut l'être si nous n'avons pas pris des précautions lors de l'écriture de la fonction "CALCULATE".

"CALCULATE" doit s'assurer que le diviseur est non nul, avant que toute opération ne puisse être effectuée. Si ce n'est pas le cas, "CALCULATE" doit imprimer un message prévenant le programmeur et laisser l'opération telle quelle.

2.1.9. Procédure Folding

On trouvera ici le texte de la procédure "FOLDING" écrite en ALGOL-JONQUILLE-74.

Le texte présenté dans ce paragraphe n'est pas celui d'une procédure optimisée. Nous n'avons pas recherché la "finesse" pour la "finesse", c'est-à-dire que nous avons tenté de conserver la "lisibilité".

```
PROCEDURE FOLDING (NOW, NV, NI, NVI); REFERENCE NOW; INTEGER NV, NI, NVI;
        COMMENT GLOBAL OPERANDS OF THE PROCEDURE ARE
                1 ACTUAL, ACTUEL  TWO VARIABLES OF TYPE REFERENCE
                2 THE STRUCTURES  TI, TSQ, TC
                3 THE INTEGERS    NC, MC, NA, MA, NPS, MPS, NIA, MIA,
                                  NCI, MCI, NAP, MAP;

BEGIN COMMENT DECLARATION OF ALL LOCAL OPERANDS;
      REFERENCE PTSTKVAL FOR STKVAL,
                PTSTKIVAL FOR STKIVAL,
                PTSTKIVVAL FOR STKIVVAL;
      STRUCTURE ITEM (INTEGER OP,
                      STRUCTURE ARG 1 (INTEGER CODE, REFERENCE POINTER),
                      STRUCTURE SUBS 1 (INTEGER CODE, REFERENCE POINTER),
                      STRUCTURE ARG 2 (INTEGER CODE, REFERENCE POINTER),
                      STRUCTURE SUBS 2 (INTEGER CODE, REFERENCE POINTER));
[1 : NV] STRUCTURE STKVAL (REFERENCE IDEN, EXCEPT (REFERENCE, STRUCTURE) VAL);
[1 : NI] STRUCTURE STKIVAL (REFERENCE IDEN, EXCEPT (REFERENCE,STRUCTURE) VAL);
```

```
[1 : NVI] STRUCTURE STKIVVAL (REFERENCE IDEN, INTEGER SUBS, EXCEPT (REFE-
          RENCE, STRUCTURE) VAL);
EXCEPT (REFERENCE, STRUCTURE) V 1, V 2, V
LOGICAL C, A 1, A 2, A 3, B 1, B 2, B 3, LOC
REFERENCE POINTER 1, POINTER 2, POINTER 3, POINTER I, IDEN

INTEGER OP, VI 1, VI 2
LABEL MEMORY TORE;
CHARACTER ATRIBUT;

PROCEDURE GENER CST (V, ATRIBUT, POINTER 3);
          EXCEPT (REFERENCE, STRUCTURE) V; CHARACTER ATRIBUT;
          REFERENCE POINTER 3;
          COMMENT TC, ACTUAL ARE GLOBAL OPERANDS;
          BEGIN ATRIBUT.TC (ACTUAL) : = ATRIBUT;
                VAL.TC (ACTUAL) : = V;
                POINTER 3 : = ACTUAL;
                ACTUAL : = ACTUAL + 1
          END;

PROCEDURE GENER II (ITEM, NOW); STRUCTURE ITEM; REFERENCE NOW;
          COMMENT TI, ACTUEL AND TSQ ARE GLOBAL OPERANDS;
          BEGIN OPER.TSQ (NOW) : = ACTUEL;
                TI (ACTUEL) : = ITEM;
                ACTUEL : = ACTUEL + 1
          END ;

LABEL PROCEDURE OPTYPE (OP); INTEGER OP;
          COMMENT NC, MC, NA, MA, NPS, MPS, NIA, MIA, NCI, MCI, NAP AND MAP
                  ARE GLOBAL OPERANDS;
          BEGIN
          IF (OP ≥ NC) ∧ (OP ≤ MC) THEN BEGIN
                                        OPTYPE : = COMPUTATIONS;
                                        GOTO RETOUR
                                        END;
          IF (OP ≤ MA) ∧ (OP ≥ NA) THEN BEGIN
                                        OPTYPE : = ASSIGNATIONS;
                                        GOTO RETOUR
                                        END;
```

```
        IF (OP ≤ MPS) ∧ (OP ≥ NPS) THEN BEGIN
                                        OPTYPE : = APPS
                                        GOTO RETOUR
                                        END;
        IF (OP ≤ MIA) ∧ (OP ≥ NIA) THEN BEGIN
                                        OPTYPE : = INDEXED ASSIGNATIONS;
                                        GOTO RETOUR
                                        END;
        IF (OP ≤ MCI) ∧ (OP ≥ NCI) THEN BEGIN
                                        OPTYPE : = CONSTANT II;
                                        GOTO RETOUR
                                        END;
        IF (OP ≤ MAP) ∧ (OP ≥ NAP) THEN OPTYPE : = AP;
RETOUR : END;
EXCEPT (REFERENCE, STRUCTURE) PROCEDURE CALCULATE (OP, V 1, V 2);
      INTEGER OP; EXCEPT (REFERENCE, STRUCTURE) V 1, V 2;
      COMMENT : CETTE PROCEDURE CALCULE LE RESULTAT DE L'OPERATION V 1 OP
                V 2. ELLE NE SERA PAS DECRITE ICI.;
     (BEGIN CALCULATE : = V 1 OP V 2  END)
PROCEDURE FIND TC (V, LOC, POINTER 3); LOGICAL LOC;
        EXCEPT (REFERENCE, STRUCTURE) V; REFERENCE POINTER 3;
        COMMENT : CETTE PROCEDURE CHERCHE S'IL EXISTE DEJA DANS LA TABLE
                  DES CONSTANTES UNE CONSTANTE DE VALEUR V. SI OUI, ALORS
                      LOC : = TRUE
                      POINTER 3 : = ADRESSE DANS TC DE CETTE CONSTANTE
                                                          SINON,
                      LOC : = FALSE
                      POINTER 3 : = 00 AU RETOUR DE LA PROCEDURE ;
CHARACTER PROCEDURE FIND ATRIBUT (OP); INTEGER OP ;
        COMMENT  CETTE FONCTION CHERCHE LE TYPE DU RESULTAT D'UNE OPERA-
                 TION INTERMEDIAIRE AYANT OP POUR OPERATEUR;
EXCEPT (REFERENCE, STRUCTURE) PROCEDURE FIND STKVAL (PTSTKVAL, F POINTER 2);
      REFERENCE PTSTKVAL FOR STKVAL; REFERENCE F POINTER 2;
      COMMENT : STKVAL IS A GLOBAL OPERAND;
```

```
BEGIN REFERENCE POINTER;
      POINTER : = PTSTKVAL;
      PTSTKVAL : = 1;
  L : IF IDEN STKVAL (PTSTKVAL) = F POINTER 2 THEN BEGIN
                                FIND STKVAL : = VAL STKVAL (PTSTKVAL);
                                PTSTKVAL : = POINTER;
                                GOTO RETOUR
                                                        END;
      PTSTKVAL : = PTSTKVAL + 1;
      IF PTSTKVAL = POINTER THEN BEGIN
                                FINDSTKVAL : = "R";
                                GOTO RETOUR
                                                        END;
      GOTO L ; RETOUR : END;
EXCEPT (REFERENCE, STRUCTURE) PROCEDURE EXECUTESTANDARD (F POINTER, V);
      REFERENCE F POINTER; EXCEPT (REFERENCE STRUCTURE) V;
      COMMENT : Cette procédure fait l'appel de la procédure standard
                (ex: Sinus, cosinus, float, etc...) en lui passant le
                paramètre V qui est une constante;
EXCEPT (REFERENCE, STRUCTURE) PROCEDURE FINDSTKIVAL (PTSTKIVAL, F POINTER 2);
      REFERENCE PTSKIVAL FOR STKIVAL; REFERENCE F POINTER 2;
      COMMENT:STKIVAL IS A GLOBAL OPERAND;
      BEGIN REFERENCE POINTER;
            POINTER : = PTSTKIVAL;
            PTSTKIVAL : = 1;
        L : IF IDEN.STKIVAL (PTSTKIVAL) = F POINTER 2 THEN BEGIN
                                FINDSTKIVAL : = VAL.STKIVAL (PTSTKIVAL);
                                PTSTKIVAL : = POINTER;
                                GOTO RETOUR
                                                        END;
        PTSTKIVAL : = PTSTKIVAL + 1;
        IF PTSTKIVAL = POINTER THEN BEGIN
                                FINDSTKIVAL : = "R";
                                GOTO RETOUR
                                                        END;
        GOTO L;
      RETOUR : END FINDSTKIVAL;
```

```
EXCEPT (REFERENCE, STRUCTURE) PROCEDURE FINDSTKIVAL (PTSTKIVVAL, F POINTER,
      V I); REFERENCE PTSTKIVVAL FOR STKIVVAL; REFERENCE F POINTER;
      EXCEPT (REFERENCE, STRUCTURE) V I;
      COMMENT : STKIVVAL IS A GLOBAL OPERAND;
      BEGIN REFERENCE POINTER;
            POINTER : = PTSTKIVVAL;
            PTSTKIVVAL : = 1;
          L : IF IDEN STKIVVAL (PTSTKIVVAL) = F POINTER THEN BEGIN
                          IF SUBS.STKIVVAL (PTSTKIVVAL) = VI THEN BEGIN
                                  FINDSTKIVVAL : = VAL.STKIVVAL(PTSTKIVVAL);
                                  PTSTKIVVAL : = POINTER;
                                  GOTO RETOUR
                                                        END
                                              END ;
          PTSTKIVVAL : = PTSTKIVVAL + 1;
          IF PTSTKIVVAL = POINTER THEN BEGIN
                                  FINDSTKIVVAL : = "R";
                                  GOTO RETOUR
                                              END;
          GOTO L;
RETOUR : END FINDSTKIVVAL;
REFERENCE PROCEDURE FINDPTSTKIVAL (PTSTKIVAL, IDEN);
      REFERENCE PTSTKIVAL FOR STKIVAL;
      REFERENCE IDEN;
      COMMENT : STKIVAL IS A GLOBAL OPERAND;
      BEGIN REFERENCE POINTER;
          POINTER : = PTSTKIVAL;
          PTSTKIVAL : = 1;
      L : IF IDEN STKIVAL (PTSTKIVAL) = IDEN THEN BEGIN
                                  FINDPTSTKIVAL : = PTSTKIVAL;
                                  PTSTKIVAL : = POINTER;
                                  GOTO RETOUR
                                                        END;
```

```
            PTSTKIVAL : = PTSTKIVAL + 1;
            IF PTSTKIVAL = POINTER THEN BEGIN
                                        FINDSTSTKIVAL : = 0;
                                        GOTO RETOUR
                                                            END;
            GOTO L;
        RETOUR : END;
REFERENCE PROCEDURE FINDPTSTKVAL (PTSTKVAL, F POINTER);
        REFERENCE PTSTKVAL FOR STKVAL;
        REFERENCE F POINTER;
        COMMENT STKVAL IS A GLOBAL OPERAND;
        BEGIN REFERENCE POINTER;
            POINTER : = PTSTKVAL;
            PTSTKVAL : = 1;
        LO  : IF IDEN.STKVAL (PTSTKVAL) = F POINTER THEN BEGIN
                                        FINDPTSTKVAL : = PTSTKVAL;
                                        PTSTKVAL : = POINTER;
                                        GOTO RETOUR
                                                            END;
            PTSTKVAL : = PTSTKVAL + 1;
            IF PTSTKVAL = POINTER THEN BEGIN
                                        FINDPTSTKVAL : = 0;
                                        GOTO RETOUR
                                                            END;
            GOTO LO;
        RETOUR : END FINDPTSTKVAL;

BEGIN COMMENT FOLD;
INITIALISATION : PTSTKVAL : = 1;
                 PTSTKIVAL: = 1;
                 PTSTKIVVAL:= 1;
NEXT ITEM : NOW : = NOW + 1;
            IF OPER.TSQ (NOW + 1) = "???" THEN GOTO RETURN;
            IDEN : = OPER.TSQ (NOW);
```

```
            ITEM : = TI (IDEN);
            OP : = OP.ITEM;
            POINTER 1 : = POI TER.ARG 1.ITEM;
            POINTER 2 : = POINTER.ARG 2.ITEM;
            GOTO OPTYPE (OP);
COMPUTATIONS : BEGIN
      IF CODE.ARG 1.ITEM = 1 THEN BEGIN
                        V 1 : = VAL.TC (POINTER 1);
                        IF CODE.ARG 2.ITEM = 1 THEN BEGIN
                              V 2 : = VAL.TC (POINTER 2);
                              GOTO CONSTANT
                                                END
                                                  ELSE
                              GOTO LEFT CONSTANT
                           END
                        ELSE BEGIN
                        IF CODE.ARG 2.ITEM = 1 THEN BEGIN
                              V 2 : = VAL.TC (POINTER 2);
                              GOTO RIGHTCONSTANT
                                                END
                                   ELSE
                              GOTO NO CONSTANT
                           END
         END COMPUTATION;
CONSTANT : BEGIN COMMENT PI : OP, CST 1, CST 2;
      V : = CALCULATE (OP, V 1, V 2);
      FIND TC (V, C, POINTER 3);
      IF ¬C THEN BEGIN COMMENT V DID NOT EXIST TILL NOW;
            ATRIBUT : = FIND ATRIBUT (OP);
            GENER CST (V, ATRIBUT, POINTER 3);
                           END;
      OP.ITEM : = 50;
      CODE.ARG 1.ITEM : = 1;
      POINTER.ARG 1.ITEM : = POINTER 3;
```

```
        ARG 2.ITEM : = OO;
        TI (IDEN) : = ITEM;
        PIT.TSQ (NOW) : = FALSE;
        GO TO NEXT ITEM
            END CONSTANT;
LEFT CONSTANT : BEGIN COMMENT PI : OP, CST 1, [VAR 2, IP 2, VAR [12]] ;
        IF CODE.ARG 2.ITEM = 2 THEN GO TO LEFT SIMPLE VAR;
        IF CODE.ARG 2.ITEM = 3 THEN GO TO LEFT SIMPLE II;
        GO TO LEFT INDEXED VAR
            END LEFT CONSTANT;
LEFT SIMPLE VAR : BEGIN COMMENT PI : OP, CST 1, VAR 2;
        V 2 : = FIND STKVAL (PTSTKVAL, POINTER 2);
        IF V 2 = "R" THEN BEGIN
                    DO.TC (POINTER 1) : = TRUE;
                    GO TO NOT FOLDABLE
                    END;
        GO TO FOLDABLE
            END LEFT SIMPLE VAR;
LEFT SIMPLE II : BEGIN COMMENT PI : OP, CST 1, PI 2;
        JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
        IF C THEN BEGIN COMMENT PT 2 = CST;
                        GO TO FOLDABLE
                    END;
        DO.TC (POINTER 1) : = TRUE;
        GO TO NOT FOLDABLE
            END LEFT SIMPLE II;
LEFT INDEXED VAR : BEGIN COMMENT PI : OP, CST 1, VAR [I2] ;
        JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT TORE;
        IF C THEN BEGIN COMMENT I2 = CST;
                        JUMP WITH RETURN ANALYSIS VAR IR, IMPLICIT TORE;
                        IF C THEN BEGIN COMMENT VAR [I2] = CST;
                                        GO TO FOLDABLE
                                END;
```

```
        IF B 3 THEN BEGIN
                    JUMP WITH RETURN REPLACE IIR, IMPLICIT  TORE;
                    GENER II (ITEM, NOW)
                    END;
        IF B 1 THEN DO.TC (POINTER I) : = TRUE
        END;
DO.TC (POINTER 1) : = TRUE;
GO TO NOT FOLDABLE
        END LEFT INDEXED VAR;
RIGHT CONSTANT : BEGIN COMMENT PI : OP, [VAR 1, PI 1, VAR [1]], CST 2;
        IF CODE.ARG 1.ITEM = 2 THEN GO TO R SIMPLE VAR;
        IF CODE.ARG 1.ITEM = 3 THEN GO TO R SIMPLE II;
        GO TO R INDEXED VAR;
            END RIGHT CONSTANT;
R SIMPLE VAR : BEGIN COMMENT PI : OP, VAR 1, CST 2;
        V 1 : = FIND STKVAL (PTSTKVAL, POINTER 1);
        IF V 1 = "R" THEN BEGIN
                          DO TC (POINTER 2) : = TRUE;
                          GO TO NOT FOLDABLE;
                          END;
        GO TO FOLDABLE
            END R SIMPLE VAR;
R SIMPLE II : BEGIN COMMENT PI : OP, PI 1, CST 2;
                    JUMP WITH RETURN ANALYSIS IIL, IMPLICIT  TORE;
                    IF C THEN BEGIN COMMENT PI 1 = CST;
                                    GO TO FOLDABLE
                          END;
                    DO.TC (POINTER 2) : = TRUE;
                    GO TO NOT FOLDABLE
            END R SIMPLE II;
R INDEXED VAR : BEGIN COMMENT PI : OP, VAR [I 1], CST 2;
        JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT  TORE;
        IF C THEN BEGIN COMMENT I 1 = CST;
                JUMP WITH RETURN ANALYSIS VAR IL, IMPLICIT  TORE;
```

```
        IF C THEN BEGIN COMMENT VAR [I 1] = CST;
                        GO TO FOLDABLE
                  END;
        IF A 3 THEN BEGIN
                        JUMP WITH RETURN REPLACE IIL, IMPLICIT  TORE;
                        GENER II (ITEM, NOW)
                        END;
        IF A 1 THEN DO.TC (POINTER 1) : = TRUE
        END I 1 = CST;
    DO.TC (POINTER 2) : = TRUE;
    GO TO NOT FOLDABLE
                        END R INDEXED VAR;
NO CONSTANT : BEGIN COMMENT PI : OP, [VAR 1, PI 1, VAR [I1] ], [VAR 2,PI2,
        IF CODE ARG 1.ITEM = 2 THEN BEGIN                          VAR [I 2]];
                        IF CODE.ARG 2.ITEM = 2 THEN GO TO ALL VARIABLES
                                               ELSE GO TO LEFT VARIABLE
                                  END;
        IF CODE ARG 2.ITEM = 2 THEN GO TO RIGHT VARIABLE
                                  ELSE GO TO NO VARIABLE
            END NO CONSTANT;
ALL VARIABLES : BEGIN COMMENT PI : OP, VAR 1, VAR 2;
                        V 1 : = FIND STKVAL (PTSTKVAL, POINTER 1);
                        IF V 1 = "R" THEN GO TO NOT FOLDABLE;
                        V 2 : = FIND STKVAL (PTSTKVAL, POINTER 2);
                        IF V 2 = "R" THEN GO TO NOT FOLDABLE
                                     ELSE GO TO FOLDABLE
            END ALL VARIABLES;
LEFT VARIABLE : BEGIN COMMENT PI : OP, VAR 1, [PI 2, VAR [I2]]
                        IF CODE ARG 2.ITEM = 2 THEN GO TO L VAR II
                                               ELSE GO TO L VAR VAR I
            END LEFT VARIABLE;
```

```
L VAR II : BEGIN COMMENT PI : OP, VAR 1, PI 2;
           JUMP WITH RETURN ANALYSIS IIR, IMPLICIT  TORE;
           IF C THEN BEGIN
               V 1 : = FIND STKVAL (PTSTKVAL, POINTER 1);
               IF V 1 = "R" THEN BEGIN
                                 JUMP WITH RETURN REPLACE IR, IMPLICIT  TORE;
                                 GENER II (ITEM, NOW);
                                 GO TO NOT FOLDABLE
                                      END;
               GO TO FOLDABLE
               END;
           GO TO NOT FOLDABLE
           END L VAR II;

L VAR VAR I : BEGIN COMMENT PI : OP, VAR 1, VAR [I2] ;
           JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT  TORE;
           IF C THEN BEGIN COMMENT I 2 = CST;
               JUMP WITH RETURN ANALYSIS VAR I R, IMPLICIT  TORE;
               IF C THEN BEGIN COMMENT VAR [I2] = CST;
                   V 1 : = FIND STKVAL (PTSTKVAL, POINTER 1);
                   IF V 1 ≠ "R" THEN GO TO FOLDABLE
                   END;
           IF B 3 THEN BEGIN
                   JUMP WITH RETURN REPLACE IIR, IMPLICIT  TORE;
                   GENER II (ITEM, NOW)
                   END
           IF B 1 THEN DO.TC (POINTER 1) : = TRUE;
               END I 2 = CST;
       GO TO NOT FOLDABLE
           END L VAR VAR I;

RIGHT VARIABLE : BEGIN COMMENT PI : OP, [PI 1, VAR I 1], VAR 2;
       IF CODE.ARG 1.ITEM = 3 THEN GO TO RII VAR
                              ELSE GO TO R VAR I VAR
           END RIGHT VARIABLE;
```

```
R VAR I VAR : BEGIN COMMENT PI : OP, VAR [I 1], VAR 2;
      JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT  TORE;
      IF C THEN BEGIN COMMENT I 1 = CST;
          JUMP WITH RETURN ANALYSIS VAR IL, IMPLICIT  TORE;
          IF C THEN BEGIN COMMENT VAR [I 1] = CST;
              V 2 : = FIND STKVAL (PTSTKVAL, POINTER 2);
              IF V 2 ≠ "R" THEN GO TO FOLDABLE
              END;
          IF A3 THEN BEGIN
              JUMP WITH RETURN REPLACE IIL, IMPLICIT  TORE;
              GENER II (ITEM, NOW)
              END
          IF A 1 THEN DO.TC (POINTER I) : = TRUE
              END I 1 = CST;
      GO TO NOT FOLDABLE
          END R VAR I VAR;

NO VARIABLE : BEGIN COMMENT PI : OP, [PI 1, VAR [I 1]], [PI 2, VAR [I 2]] ;
      IF CODE.ARG 1.ITEM = 3 THEN BEGIN
              IF CODE.ARG 2.ITEM = 3 THEN GO TO ALL II
                                     ELSE GO TO LII VAR I
                              END;
      IF CODE.ARG 2.ITEM = 3 THEN GO TO R VAR III
                              ELSE GO TO ALL VAR I
          END NO VARIABLE;

ALL II : BEGIN COMMENT PI : OP; PI 1, PI 2;
      JUMP WITH RETURN ANALYSIS IIL, IMPLICIT  TORE;
      IF C THEN BEGIN
              JUMP WITH RETURN ANALYSIS IIR, IMPLICIT  TORE;
              IF C THEN GO TO FOLDABLE;
              JUMP WITH RETURN REPLACE IL, IMPLICIT  TORE;
              GENER II (ITEM, NOW);
              GO TO NOT FOLDABLE
              END;
```

```
        JUMP WITH RETURN ANALYSIS IIR, IMPLICIT  TORE;
        IF C THEN BEGIN
                JUMP WITH RETUR  REPLACE IR, IMPLICIT  TORE;
                GENER II (ITEM, NOW);
                END;
        GO TO NOT FOLDABLE
        END ALL II;
LII VAR I : BEGIN COMMENT PI : OP, PI 1, VAR [I 2] ;
        JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT  TORE;
        IF C THEN BEGIN COMMENT I C = CST;
                        JUMP WITH RETURN ANALYSIS VAR IR, IMPLICIT  TORE;
                        IF C THEN BEGIN COMMENT VAR [I 2] = CST;
                                JUMP WITH RETURN ANALYSIS  IIL,  IMPLICIT TORE;
                                IF C THEN BEGIN COMMENT (PI:PI1,VAR [I2] )= CST;
                                        GO TO FOLDABLE
                                END;
                        IF B 3 THEN BEGIN
                                JUMP WITH RETURN REPLACE IIR, IMPLICIT TORE;
                                GENER II (ITEM, NOW)
                                END;
                        IF B 1 THEN DO TC (POINTER I) : = TRUE;
                        GO TO NOT FOLDABLE
                        END VAR [I2] = CST;
        JUMP WITH RETURN ANALYSIS IIL, IMPLICIT TORE;
        IF B 3 V C THEN BEGIN
                        IF B 3 THEN JUMP WITH RETURN REPLACE IIR,IMPLICIT TORE;
                        IF C THEN JUMP WITH RETURN REPLACE IL, IMPLICIT TORE;
                        GENER II (ITEM, NOW)
                        END;
        IF B 1 THEN DO.TC (POINTER I):= TRUE;
        GO TO NOT FOLDABLE
    END I 2 = CST;
```

```
        JUMP WITH RETURN ANALYSIS IIL, IMPLICIT TORE;
        IF C THEN BEGIN JUMP WITH RETURN REPLACE IL, IMPLICIT TORE;
                        GENER II (ITEM, NOW)
                  END;
        GO TO NOT FOLDABLE
           END LII VAR I;
R VAR III : BEGIN COMMENT PI : OP, VAR [I 1] , PI 2;
        JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
        IF C THEN BEGIN COMMENT I 1 = CST;
                  JUMP WITH RETURN ANALYSIS VAR IL, IMPLICIT TORE;
                  IF C THEN BEGIN COMMENT VAR [I 1] = CST;
                            JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
                            IF C THEN BEGIN COMMENT (PI:OP,VAR [I 1],PI 2)=CST;
                                           GO TO FOLDABLE
                                      END;
                            IF A 3 THEN BEGIN
                                   JUMP WITH RETURN REPLACE IIL,IMPLICIT TORE;
                                   GENER II (NOW, ITEM)
                                      END;
                            IF A 1 THEN DO.TC (POINTER I) : = TRUE;
                            GO TO NOT FOLDABLE
                            END VAR [I 1] = CST;
                  JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
                  IF A 3 V C THEN BEGIN
                            IF A 3 THEN JUMP WITH RETURN REPLACE IIL,
                                                         IMPLICIT TORE;
                            IF C THEN JUMP WITH RETURN REPLACE IL,
                                                         IMPLICIT TORE;
                            GENER II (ITEM, NOW)
                            END;
                  IF A 1 THEN DO TC (POINTER I) : = TRUE;
                  GO TO NOT FOLDABLE
                  END I 1 = CST;
```

```
        JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
        IF C THEN BEGIN JUMP WITH RETURN REPLACE IR, IMPLICIT TORE;
                GENER II (ITEM, N W)
            END;
        GO TO NOT FOLDABLE
            END R VAR III;

ALL VAR I : BEGIN COMMENT PI : OP, VAR [I 1] , VAR [I 2] ;
        JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT TORE;
        IF C THEN BEGIN COMMENT I 2 = CST;
                JUMP WITH RETURN ANALYSIS VAR IR, IMPLICIT TORE;
                IF C THEN BEGIN COMMENT VAR [I 2] = CST;
                  JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
                  IF C THEN BEGIN COMMENT (VAR [I 2] ∧ I 1) = CST;
                       JUMP WITH RETURN ANALYSIS VAR IL, IMPLICIT TORE;
                       IF C THEN BEGIN COMMENT (PI:OP,VAR [I 1],VAR [I 2])
                                                                  = CST;
                               GO TO FOLDABLE
                               END;
                       IF A 3 V B 3 THEN BEGIN
                               IF A 3 THEN JUMP WITH RETURN REPLACE IIL,
                                                        IMPLICIT TORE;
                               IF B 3 THEN JUMP WITH RETURN REPLACE IIR;
                                                        IMPLICIT TORE;
                               GENER II (ITEM, NOW)
                                     END;
                       IF A 1 THEN DO.TC (POINTER.SUBS 1.ITEM): = TRUE;
                       IF B 1 THEN DO.TC (POINTER.SUBS 2.ITEM): = TRUE;
                       GO TO NOT FOLDABLE
                               END (VAR [I 2] ∧ I 1) = CST;
                       IF A 3 THEN BEGIN
                             JUMP WITH RETURN REPLACE IIL, IMPLICIT TORE;
                             GENER II (ITEM, NOW)
                                     END;
                       IF A 1 THEN DO.TC (POINTER.SUBS 1.ITEM): = TRUE;
                       GO TO NOT FOLDABLE
                             END VAR [I 2] = CST;
```

```
        JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
        IF A 3 V B 3 THEN BEGIN
                              IF A 3 THEN JUMP WITH RETURN REPLACE IIL,
                                                           IMPLICIT TORE;
                              IF B 3 THEN JUMP WITH RETURN REPLACE IIR,
                                                           IMPLICIT TORE;
                              GENER II (ITEM, NOW)
                              END;
        IF A 1 THEN DO.TC (POINTER.SUBS 1.ITEM): = TRUE;
        IF B 1 THEN DO.TC (POINTER.SUBS 2.ITEM): = TRUE;
        GO TO NOT FOLDABLE
        END I 2 = CST;
    JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
    IF A 3 THEN BEGIN JUMP WITH RETURN REPLACE IIL, IMPLICIT TORE;
                      GENER II (ITEM, NOW);
                      GO TO NOT FOLDABLE
        END;
    IF A 1 THEN DO.TC (POINTER.SUBS 1.ITEM): = TRUE;
    GO TO NOT FOLDABLE
        END ALL VAR I;
    END COMPUTATION;

ASSIGNATIONS : BEGIN COMMENT PI: : =, VAR 1, [CST 2, PI 2, VAR 2, VAR [I 2]];
    IF CODE.ARG 2.ITEM = 1 THEN BEGIN
                              V 2: = VAL.TC (POINTER 2);
                              DO.TC (POINTER 2): = TRUE;
                              GO TO FOLDABLE ASS
                                 END
    IF CODE.ARG 2.ITEM = 2 THEN BEGIN
                              V 2: = FIND STKVAL (PTSTKVAL, POINTER 2);
                              IF V 2 = "R" THEN GO TO NOT FOLDABLE ASS
                                          ELSE GO TO FOLDABLE ASS
                                 END;
    IF CODE.ARG 2.ITEM = 3 THEN BEGIN
                 JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
```

```
            IF C THEN BEGIN
                JUMP WITH RETURN REPLACE IR, IMPLICIT TORE;
                GENER II (IDEN, NOW);
                GO TO FOLDABLE ASS
                        END;
            GO TO NOT FOLDABLE ASS
                        END;
        JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT TORE;
            IF C THEN BEGIN COMMENT I 2 = CST;
                    IF B 3 THEN BEGIN
                                JUMP WITH RETURN REPLACE IIR,
                                                 IMPLICIT TORE;
                                GENER II (ITEM, NOW)
                                END;
            IF B 1 THEN DO.TC (POINTER I): = TRUE ;
            JUMP WITH RETURN ANALYSIS VAR IR, IMPLICIT TORE;
            IF C THEN BEGIN COMMENT VAR [I 2] = CST;
                        GO TO FOLDABLE ASS
                    END
            END;
        GO TO NOT FOLDABLE ASS
            END ASSIGNATIONS;

INDEXED ASSIGNATIONS : BEGIN COMMENT PI: : =, VAR [I 1],[PI 2,CST 2,VAR 2,
                                                         VAR [I 2] ];
    JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
    IF C THEN BEGIN COMMENT I 1 = CST;
            IF A 1 THEN BEGIN
                    DO.TC (POINTER I): = TRUE;
                    GO TO ANAL ARG 2
                    END ;
            IF A 3 THEN JUMP WITH RETURN REPLACE IIL, IMPLICIT TORE;
            ANALARG 2 : BEGIN
                    IF CODE.ARG 2.ITEM= 1 THEN BEGIN COMMENT CST 2;
```

```
                    IF A 3 THEN GENER II (ITEM, NOW);
                    DO.TC (POINTER 2): = TRUE;
                    GO TO FOLDABLE IASS
                          END CST 2;
          IF CODE.ARG 2.ITEM = 2 THEN BEGIN COMMENT V 2;
                    IF A 3 THEN GENER II (ITEM, NOW);
                    V 2: = FIND STKVAL (PTSTKVAL, POINTER 2)
                    IF V 2 = "R" THEN GO TO NOT FOLDABLE IASS
                                 ELSE GO TO FOLDABLE IASS
                          END V 2;
          IF CODE.ARG 2.ITEM = 3 THEN BEGIN COMMENT PI 2;
                    JUMP WITH RETURN ANALYSIS IIR,IMPLICIT TORE;
                    IF C THEN BEGIN COMMENT PI 2 = CST;
                         JUMP WITH RETURN REPLACE IR,
                                          IMPLICIT TORE;
                         GENER II (ITEM, NOW);
                         GO TO FOLDABLE IASS
                           END PI 2 = CST;
                    IF A 3 THEN GENER II (ITEM, NOW);
                    GO TO NOT FOLDABLE IASS
                          END PI 2;
                    JUMP WITH RETURN ANALYSIS R INDEX,
                                          IMPLICIT TORE;
                    IF C THEN BEGIN COMMENT (I 2= CST)∧(I1 =CST);
                          IF B 1 THEN BEGIN
                                 DO.TC(POINTER I): = TRUE;
                                 GO TO ANAL VAR I 2
                                 END
                    IF B 3 THEN JUMP WITH RETURN REPLACE IIR,
                                          IMPLICIT TORE;
ANAL VAR I2: IF A 3 V B 3 THEN GENER II (ITEM, NOW);
     JUMP WITH RETURN ANALYSIS VAR R, IMPLICIT TORE;
     IF C THEN GO TO FOLDABLE IASS
          ELSE GO TO NOT FOLDABLE IASS
          END (I 2 = CST) ∧ (I 1 = CST);
```

```
            IF A 3 THEN GENER II (ITEM, NOW);
            GO TO NOT FOLDABLE IASS
        END I 1 = CST;

JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT TORE;
IF C THEN BEGIN COMMENT (I 2 = CST) ∧ (I 1 ≠ CST);
        IF B 1 THEN BEGIN
                    DO.TC (POINTER I): = TRUE;
                    GO TO NOT FOLDABLE IASS
                    END;
        IF B 3 THEN BEGIN
                    JUMP WITH RETURN REPLACE IIR, IMPLICIT TORE;
                    GENER II (ITEM, NOW)
                    END
        END (I 2 = CST) ∧ (I 1 ≠ CST);
GO TO NOT FOLDABLE IASS
      END INDEXED ASSIGNATIONS;

CONSTANT II : BEGIN COMMENT PI : 50, CST 1, 00;
                BIT.TSQ (NOW): = FALSE;
                GO TO NEXT ITEM
            END CONSTANT II;

APPS : BEGIN COMMENT PI: APS, PS, [CST 2, VAR 2, PI 2, VAR [I 2]];
      IF CODE.ARG 2.ITEM = 1 THEN BEGIN
                             V 2 : = VAL.TC (POINTER 2);
                             GO TO FOLDABLE APS
                                END;
      IF CODE.ARG 2.ITEM = 2 THEN BEGIN
                             V 2 : = FIND STKVAL (PTSTKVAL, POINTER 2);
                             IF V 2 = "R" THEN GO TO NOT FOLDABLE
                                          ELSE GO TO FOLDABLE APS
                                END;
      IF CODE.ARG 2.ITEM = 3 THEN BEGIN
                             JUMP WITH RETURN ANALYSIS IIR, IMPLICIT TORE;
                             IF C THEN GO TO FOLDABLE APS
                                  ELSE GO TO NOT FOLDABLE
                                END;
```

```
JUMP WITH RETURN ANALYSIS R INDEX, IMPLICIT TORE;
IF C THEN BEGIN
        JUMP WITH RETURN ANALYSIS VAR IR, IMPLICIT TORE;
        IF C THEN GO TO FOLDABLE ADS;
        IF B 1 THEN DO.TC (POINTER I): = TRUE;
        IF B 3 THEN BEGIN
                JUMP WITH RETURN REPLACE IIR, IMPLICIT TORE;
                GENER II (ITEM, NOW)
                END
        END;
GO TO NOT FOLDABLE
      END APPS;

AP : BEGIN COMMENT Cette partie sera traitée en détail dans le texte écrit,
           où l'on envisagera ce qu'il est nécessaire de faire sui-
           vant que l'on optimise un programme ALGOL, FORTRAN ou
           PL/1. La séquence AP ne sera pas programmée dans notre
           langage ALGOL-JONQUILLE-74;
END AP;

FOLDABLE : BEGIN COMMENT FOLDING COMPUTATIONS;
        V: = CALCULATE (OP, V 1, V 2);
        BIT.TSQ (NOW): = FALSE;
        JUMP WITH RETURN MODIFY STKIVAL, IMPLICIT TORE;
        GO TO NEXT ITEM;
        END FOLDABLE;

NOT FOLDABLE : BEGIN COMMENT (OP, ARG 1, ARG 2) ≠ CST;
        JUMP WITH RETURN SUPRESS TKIVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
        END NOT FOLDABLE;

MODIFY STKIVAL : BEGIN COMMENT THIS SEQUENCE MODIFIES THE VALUE OF AN EXIS-
           TING ENTRY IN STKIVAL OR ADDS A NEW ENTRY TO STKIVAL;
           POINTER 3: = FIND PTSTKIVAL (PTSTKIVAL, IDEN);
```

```
            IF POINTER 3 = 0 THEN BEGIN
                                IDEN STKIVAL (PTSTKIVAL): = IDEN;
                                POINTER 3: = PTSTKIVAL,
                                PTSTKIVAL: = PTSTKIVAL + 1
                                     END;
            VAL.STKIVAL (POINTER 3): = V;
            JUMP BACK TORE;
            END MODIFY STKIVAL;

SUPRESS STKIVAL : BEGIN COMMENT THIS SEQUENCE WILL DELETE AN EXISTING ENTRY
                              IN STKIVAL;
               POINTER 3: = FIND PTSTKIVAL (PTSTKIVAL, IDEN);
               IF POINTER 3 = 0 THEN JUMP BACK TORE;
                  POINTER I1: = PTSTKIVAL - 1;
                  PTSTKIVAL: = POINTER 3;
            LO :  IF PTSTKIVAL = POINTERI1 THEN JUMP BACK TORE;
                  STKIVAL (PTSTKIVAL): = STKIVAL (PTSTKIVAL + 1);
                  PTSTKIVAL: = PTSTKIVAL + 1;
                  GO TO LO;
            END SUPRESS STKIVAL;

FOLDABLE ASS : BEGIN COMMENT PI: : =, VAR, CST;
        JUMP WITH RETURN MODIFY STKVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
              END FOLDABLE ASS;

NOT FOLDABLE ASS : BEGIN COMMENT
        JUMP WITH RETURN SUPRESS STKVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
              END NOT FOLDABLE ASS;

MODIFY STKVAL : BEGIN COMMENT THIS SEQUENCE IS SIMILAR TO THE SEQUENCE
                              MODIFY STKIVAL;
      POINTER 3: = FIND PTSTKVAL (PTSTKVAL, POINTER 1);
```

```
        IF POINTER 3 = 0 THEN BEGIN
                                IDEN STKVAL (PTSTKVAL): = POINTER 1;
                                POINTER 3: = PTSTKVAL;
                                PTSTKVAL: = PTSTKVAL + 1
                                        END;
        VAL.STKVAL (POINTER 3): = V;
        JUMP BACK TORE;
            END MODIFY STKVAL;

SUPRESS STKVAL : BEGIN COMMENT THIS SEQUENCE IS SIMILAR TO SUPRESS STKIVAL;
        POINTER 3: = FIND PTSTKVAL (PTSTKVAL, POINTER 1);
        IF POINTER 3 = 0 THEN JUMP BACK TORE;
        POINTER I 1: = PTSTKVAL - 1;
        PTSTKVAL: = POINTER 3;
   LO :IF PTSTKVAL = POINTER I 1 THEN JUMP BACK TORE;
        STKVAL (PTSTKVAL): = STKVAL (PTSTKVAL + 1);
        PTSTKVAL: = PTSTKVAL + 1;
        GO TO LO
   END SUPRESS STKVAL;

FOLDABLE IASS : BEGIN
        JUMP WITH RETURN MODIFY STKVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
            END;

NOT FOLDABLE IASS : BEGIN
        JUMP WITH RETURN SUPRESS STKIVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
            END;

MODIFY STKIVVAL : BEGIN COMMENT MODIFY STKIVVAL IS SIMILAR TO MODIFY STKIVAL
                                AND STKVAL;
        POINTER I 2: = PTSTKIVVAL;
        PTSTKIVVAL: = 1;
```

```
LO : IF IDEN.STKIVVAL (PTSTKIVVAL) = POINTER 1 THEN BEGIN
          IF SUBS.STKIVVAL (PTSTKIVVAL) = VI 1 THEN BEGIN
                                VAL.STKIVVAL (PTSTKIVVAL): = V 2;
     PTSTKIVVAL: = POINTER I 2; JUMP BACK TORE
                                                END
                                     END;
     PTSTKIVVAL: = PTSTKIVVAL + 1;
     IF PTSTKIVVAL = POINTER I 2 THEN BEGIN
                                IDEN.STKIVVAL (PTSTKIVVAL): = POINTER 1;
                                SUBS.STKIVVAL (PTSTKIVVAL): = VI 1;
                                VAL.STKIVVAL (PTSTKIVVAL):  = V 2;
                                PTSTKIVVAL: = PTSTKIVVAL + 1;
                                JUMP BACK TORE
                                     END;
     GO TO LO
        END MODIFY STKIVVAL;

SUPRESS STKIVVAL : BEGIN COMMENT THIS SEQUENCE IS DIFFERENT FROM THE OTHER
                                SUPPRESS SEQUENCES;
     POINTER I 2: = PTSTKIVVAL; C: = FALSE;
     PTSTKIVVAL: = 1;
  L 1: IF IDEN.STKIVVAL (PTSTKIVVAL) = POINTER 1 THEN BEGIN
                                IDEN.STKIVVAL (PTSTKIVVAL):= OO;C:= CV TRUE
                                                END
     PTSTKIVVAL: = PTSTKIVVAL + 1;
     IF PTSTKIVVAL = POINTER I 2 THEN GO TO COMPACT
                                  ELSE GO TO L 1;
COMPACT : IF ¬ C THEN JUMP BACK TORE;
        PTSTKIVVAL: = 1;
  L 2 : IF IDEN.STKIVVAL.(PTSTKIVVAL) ≠ OO THEN BEGIN
                                PTSTKIVVAL: = PTSTKIVVAL + 1;
                                GO TO L 2
                                                END;
```

```
BEGIN REFERENCE POINTER 4 FOR STKIVVAL;
    POINTER 4: = PTSTKIVVAL;
L 3:PTSTKIVVAL: = PTSTKIV AL + 1;
    IF PTSTKIVVAL = POINTER I 2 THEN BEGIN
                                    PTSTKIVVAL: = POINTER 4;
                                    JUMP BACK TORE
                                            END;
    IF IDEN.STKIVVAL (PTSTKIVVAL) = 00 THEN GO TO L 3;
    STKIVVAL (POINTER 4): = STKIVVAL (PTSTKIVVAL);
    POINTER 4: = POINTER 4 + 1;
    GO TO L 3
END
        END SUPPRESS STKIVVAL;

FOLDABLE APS : BEGIN
        V: = EXECUTE STANDARD (POINTER 1, V 2);
        BIT.TSQ (NOW): = FALSE;
        JUMP WITH RETURN MODIFY STKIVAL, IMPLICIT TORE;
        GO TO NEXT ITEM
END FOLDABLE APS;

ANALYSIS L INDEX : BEGIN COMMENT ANALYSE DE I 1 DANS VAR [I 1] ;
    POINTER I 1: = POINTER SU 3 1.ITEM;
    A 1: = A 2: = A 3: = FALSE;
    IF CODE SUBS 1.ITEM = 1 THEN BEGIN C: = TRUE;
                                    A1: = TRUE;
                                VI 1: = VAL.TC (POINTER I);
                                JUMP BACK TORE
                            END;
    IF CODE SUBS 1.ITEM = 2 THEN BEGIN
            VI 1: = FIND STKVAL (PTSTKVAL, POINTER I 1);
            IF VI 1 = "R" THEN BEGIN C: = FALSE;
                                JUMP BACK TORE
                            END
```

```
                    ELSE BEGIN  C: = TRUE;
                                A2: = TRUE;
                                JUMP BACK TORE
                         END
                END;
VI 1: = IF OP.TI (POINTER I 1) = 50 THEN VAL.TC (POINTER.ARG 1.TI(POINTER I 1)
                                    ELSE FIND STKIVAL (PTSTKIVAL, POINTER I 1);
IF VI 1 = "R" THEN BEGIN  C: = FALSE;
                          JUMP BACK TORE
                END
                ELSE BEGIN  C: = TRUE; A 3: = TRUE;
                          JUMP BACK TORE
                END
            END ANALYSIS L INDEX;

ANALYSIS R INDEX : BEGIN COMMENT ANALYSE DE I 2 DANS VAR [I 2] ;
        B 1: = B 2: = B 3: = FALSE;
        POINTER I 2: = POINTER.SUBS 2.ITEM;
        IF CODE.SUBS 2.ITEM = 1 THEN BEGIN  C: = TRUE; B 1: = TRUE;
                          V I 2: = VAL TC (POINTER I); JUMP BACK TORE
                                        END;
        IF CODE.SUBS 2.ITEM = 2 THEN BEGIN
                          V I 2: = FIND STKVAL (PTSTKVAL, POINTER I 2)
                          IF V I 2 = "R" THEN BEGIN  C: = FALSE;
                                                JUMP BACK TORE
                                        END
                          ELSE BEGIN  C: = B 2: = TRUE;
                                      JUMP BACK TORE
                                        END
                          END;
V I 2: = IF OP.TI (POINTER I 2) = 50 THEN VAL.TC(POINTER.ARG 1.TI(POINTER I2))
                                    ELSE FIND STKIVAL (PTSTKIVAL,POINTER I2);
IF V I 2 = "R" THEN BEGIN  C: = FALSE; JUMP BACK TORE  END
               ELSE BEGIN  C: = B 3: = TRUE; JUMP BACK TORE  END
               END ANALYSIS R INDEX;
```

```
ANALYSIS IIR : BEGIN COMMENT ANALYSE D'UNE II SITUEE A DROITE;
      V 2: = IF OP.II(POINTER 2)= 50 THEN VAL.TC(POINTER.ARG 1.II(POINTER 2))
                                     ELSE FIND STKIVAL(PTSTKIVAL,POINTER 2);
      IF V 2 = "R" THEN  C: = FALSE
                   ELSE  C: = TRUE;
      JUMP BACK TORE
          END ANALYSIS IIR;

ANALYSIS IIL : BEGIN COMMENT ANALYSE D'UNE II SITUEE A GAUCHE;
      V 1: = IF OP.TI(POINTER 1)= 50 THEN VAL.TC(POINTER.ARG 1.II(POINTER 1))
                                     ELSE FIND STKIVAL(PTSTKIVAL,POINTER 1)
      IF V 1 = "R" THEN  C: = FALSE
                   ELSE  C: = TRUE;
      JUMP BACK TORE
          END ANALYSIS IIL;

ANALYSIS VAR IR : BEGIN
      V 2: = FIND STKIVVAL (PTSTKIVVAL, POINTER 2, VI 2);
      IF V 2 = "R" THEN  C: = FALSE
                   ELSE  C: = TRUE;
      JUMP BACK TORE
          END ANALYSIS VAR IR;

ANALYSIS VAR IL : BEGIN
      V 1: = FIND STKIVVAL (PTSTKIVVAL, POINTER 1, VI 1);
      IF V 1 = "R" THEN  C: = FALSE
                   ELSE  C: = TRUE;
      JUMPBACK TORE
          END ANALYSIS VAR IL;

REPLACE IIL : BEGIN COMMENT REPLACING OF PI 1 IN VAR [PI 1] ;
      IF OP.TI(POINTER 1) = 50 THEN POINTER 3: = POINTER.ARG 1.TI(POINTER I1)
                               ELSE BEGIN
                 FIND TC (VI 1, LOC, POINTER 3);
                 IF ¬ LOC THEN BEGIN
```

```
                                ATRIBUT: = "I";
                                GENER CST (VI 1, ATRIBUT, POINTER 3)
                                          END
                              END;
        DO.TC (POINTER 3): = TRUE; CODE.SUBS 1.ITEM: = 1;
        POINTER.SUBS 1.ITEM: = POINTER 3;
        JUMP BACK TORE
            END REPLACE IIL;

REPLACE IIR : BEGIN COMMENT REPLACING OF PI2 IN VAR [PI2] ;
        IF OP.TI (POINTERI2) = 60 THEN POINTER 3:= POINTER.ARG 1.TI(POINTERI2)
                                ELSE BEGIN
                FIND TC(VI2, LOC, POINTER 3);
                IF ¬ LOC THEN BEGIN
                                ATRIBUT: = "I"
                                GENER CST (VI2,ATRIBUT,POINTER 3)
                                          END
                              END ELSE;
        DO.TC (POINTER 3): = TRUE;
        CODE.SUBS 2.ITEM: = 1;
        POINTER.SUBS 2.ITEM: = POINTER 3;
        JUMP BACK TORE
              END REPLACE IIR;

REPLACE IR : BEGIN COMMENT REPLACING OF PI2 BY A CONSTANT;
        IF OP.TI (POINTER 2) = 50 THEN POINTER 3: = POINTER.ARG 1.TI(POINTER 2)
                                ELSE BEGIN
                FIND TC (V2, LOC, POINTER 3);
                IF   LOC THEN BEGIN
                                ATRIBUT: = FIND ATRIBUT (OP);
                                GENER CST (V2, ATRIBUT, POINTER 3)
                                          END
                              END;
        DO.TC (POINTER 3): = TRUE;
        CODE.ARG 2.ITEM: = 1;
```

```
        POINTER.ARG 2.ITEM: = POINTER 3;
        JUMP BACK TORE
            END REPLACE IR;

REPLACE IL : BEGIN COMMENT REPLACING OF PI 1 BY A CONSTANT;
        IF OP.TI (POINTER 1) = 50 THEN POINTER 3: = POINTER.ARG 1.TI(POINTER 1)
                                  ELSE BEGIN;
                                  FIND TC(V 1, LOC, POINTER 3);
                                  IF   LOC THEN BEGIN
                                       ATRIBUT: = FIND ATRIBUT (OP);
                                       GENER CST (V2, ATRIBUT, POINTER 3)
                                                    END
                                     END;
        DO.TC (POINTER 3): = TRUE;
        CODE.ARG 1.ITEM: = 1;
        POINTER.ARG 1.ITEM: = POINTER 3;
        JUMP BACK TORE
            END REPLACE IL;
```

RETURN BEGIN COMMENT. Nous supposons ici nous situer dans le cadre d'un compilateur optimiseur pour un langage source du type FORTRAN ou ALGOL (sauf que dès lors nous n'admettrions pas des procédures ayant parmi leurs paramètres des procédures, des étiquettes ou des "switchs". Le pourquoi de ceci sera expliqué dans le texte, chapitre II). Dès lors, la seule opération intermédiaire pouvant terminer un segment d'instructions est un branchement.

Rappel du format général d'un brahcement qui a pour opérande une valeur :

PI : OP; ARG 1, POINTER TO TDS.

Nous détaillerons dans le texte ce que l'on pourrait envisager comme traitement plus raffiné pour ce genre d'opération intermédiaire;

```
IF CODE.ARG 1.ITEM = 3 THEN BEGIN
                       JUMP WITH RETURN ANALYSIS IIL, IMPLICIT TORE;
                       IF C THEN BEGIN
                            JUMP WITH RETURN REPLACE IL, IMPLICIT TORE;
                            GENER II (ITEM, NOW - 1);
                            GO TO RETOUR
                                 END;
                       GO TO RETOUR
                            END;
IF CODE.ARG 1.ITEM = 4 THEN BEGIN
                       JUMP WITH RETURN ANALYSIS L INDEX, IMPLICIT TORE;
                       IF A 1 THEN DO.TC (POINTER I1): = TRUE;
                       IF A 3 THEN BEGIN
                            JUMP WITH RETURN REPLACE IIL, IMPLICIT TORE;
                            GENER II (ITEM, NOW - 1)
                            GO TO RETOUR
                                 END;
                            END  END FOLD;
          RETOUR : END PROCEDURE FOLDING;
```

x

x x

2.2. Suppression d'instructions redondantes

Dans ce paragraphe, nous étudierons la suppression d'instructions redondantes dans un segment d'instructions.

Nous donnerons une définition générale de la notion d'instructions redondantes et nous mettrons en évidence la difficulté de son application dans sa généralité. Nous montrerons à cette occasion que l'application, dans sa généralité, de la suppression d'instructions redondantes peut conduire à un programme moins performant.

Nous donnerons ensuite un algorithme réalisant l'élimination d'instructions redondantes à l'intérieur d'un segment d'instructions. Nous critiquerons également la nécessité de la suppression d'instructions redondantes.

Les références de base sont l'ouvrage de Gries [GRIES 71], l'article d'Allen [ALLEN 69] et aussi l'ouvrage de Cocke [COCKE 70] mais dans une moindre mesure.

2.2.1. Définition de la notion d'instructions redondantes

Avant de définir la notion d'instruction redondante, nous introduirons la notion d'identité formelle entre deux instructions intermédiaires.

Définition D_1

Deux instructions intermédiaires I 1 et I 2 seront dites formellement identiques si elles ont même opérateur et mêmes opérandes.

Exemple (1) APS; sin; 3.14159265 et APS; sin; 3.14159265

(2) * ; A; B et * ; A; B

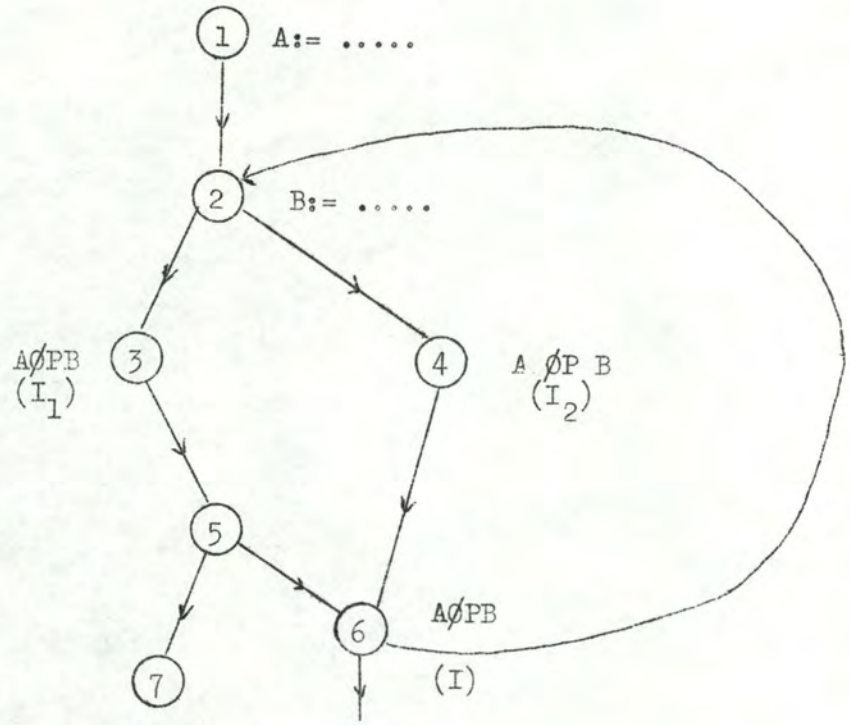
Définition D_2

Une instruction intermédiaire I est dite "redondante" si et seulement si

1° il existe une suite $\mathcal{F} = (I_1, \ldots, I_n)$ d'instructions formellement identiques à I.

2° Quel que soit un chemin allant de l'entrée du programme ver I, il existe, sur ce chemin, une instruction $I_i \in \mathcal{F}$ telle que, sur le reste du chemin allant de I_i à I, on ne trouve pas d'instruction modifiant la valeur d'un opérande de I.

Exemple



Si nous supposons qu'il n'y a pas d'instructions modifiant les valeurs de A et de B sur les chemins (3, 5, 6) et (4, 6), l'instruction I est redondante en vertu de la définition D2.

2.2.2. D3. Définition de la notion "Elimination d'instructions redondantes" (suppression)

La "Suppression d'instructions redondantes" est le processus P4 qui supprime du programme en forme intermédiaire les instructions satisfaisant à la définition D_2 et qui remplace toute référence à une instruction redondante par une référence à l'instruction formellement identique non redondante la plus proche de l'instruction redondante référencée.

2.2.3. Critique des définitions D_2 et D_3

De la définition D2 nous pouvons déduire l'algorithme suivant qui décide si une instruction I est redondante ou non.

Soit l'instruction I se trouvant dans un segment S.

(0) Déterminer si I est précédée, dans le segment S, par des instructions modifiant la valeur d'un opérande de I.

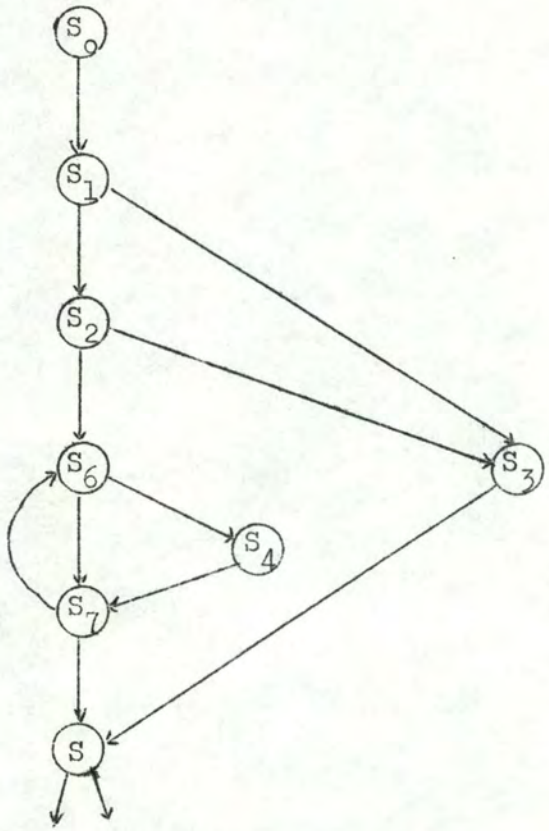
<u>si</u> <u>oui</u> <u>alors</u> terminer l'algorithme avec la réponse "I n'est pas redondante".

<u>sinon</u> poursuivre en (1)

(1) Trouver l'ensemble $\mathcal{U} = \{ u_1, u_2, \ldots, u_n \}$ des chemins distincts qui vont de l'entrée du programme (S_o) vers le segment S, chaque chemin u_i étant tel qu'il ne présente pas de sous-séquences (de sommets) adjacentes identiques. Par cette dernière restriction, nous voulons éviter que des chemins qui traversent des boucles énumèrent plusieurs fois les segments qui constituent le tout ou partie de la boucle. Sans cette restriction, la liste $\mathcal{U}$ peut très bien être non finie.

<u>Exemple</u>

1) Les chemins $u_i = \{ S_o, S_1, S_2, S_6, S_7, S \}$ et $u_j = \{ S_o, S_1, S_2, S_6, S_7, S_6, S_7, S \}$ seront considérés comme étant identiques.



Pour ce graphe, la liste $\mathcal{U}$ est la suivante

$$\{(S_o,S_1,S_3,S),\ (S_o,S_1,S_2,S_3,S),\ (S_o,S_1,S_2,S_6,S_7,S),\ (S_o,S_1,S_2,S_6,S_4,S_7,S),\ (S_o,S_1,S_2,S_6,S_4,S_7,S_6,S_7,S),\ (S_o,S_1,S_2,S_6,S_7,S_6,S_4,S_7,S)\}$$

Tout autre chemin peut être réduit à l'une de ses formes par suppression des sous-séquences adjacentes identiques.

(2) i:= 1;

(3) + Prendre en considération le chemin $u_i = \{ S_o,S_1^i,\ldots,S_m^i,S \}$ de la liste $\mathcal{U}$;

+ Déterminer s'il existe sur ce chemin u_i une instruction I_i formellement identique à I;

+ <u>Si</u> non <u>Alors</u> terminer l'algorithme avec la réponse "I n'est pas redondante"

<u>Sinon</u> <u>Début</u> 1;

- Déterminer s'il n'existe pas sur u_i une autre instruction I_i formellement identique à I et telle que la longueur* du "sous-chemin" de u_i qui sépare I_i de I soit minimale.
  Soit I_i donc l'instruction formellement identique à I et supposons que $I_i \in S_j^i$ $\quad 1 \leq j \leq m$
- Déterminer si, dans les instructions $\in S_j^i$ qui suivent l'instruction I_i, il existe une ou plusieurs instructions modifiant la valeur d'un opérande de I.
- <u>Si</u> oui <u>Alors</u> terminer l'algorithme avec la réponse "I n'est pas redondante"

  <u>Sinon</u> <u>Début</u> 2;
- Déterminer s'il existe sur le chemin $\{ S_{j+1}^i,S_{j+2}^i,\ldots,S_m^i \}$, une ou plusieurs instructions modifiant la valeur d'un opérande de I;

  <u>Si</u> oui <u>Alors</u> terminer l'algorithme avec la réponse "I n'est pas redondante"

  <u>Sinon</u> <u>Début</u> 3;

  i:= i+1;

---

* Longueur d'un chemin est le nombre d'arcs qui composent le chemin.

Si la liste $\mathcal{U}$ est épuisée
Alors terminer l'algorithme avec la réponse "I est redondante"
Sinon poursuivre en (3)
Fin 3;

Fin 2;

Fin 1;

Cet algorithme est la traduction textuelle de la définition D2. Nous remarquons qu'il n'est pas raisonnable d'envisager une application de cet algorithme pour les raisons suivantes.

1° Il est nécessaire d'énumérer tous les chemins dont parle la définition, ce qui est relativement long.

2° La détermination de l'instruction formellement identique à l'instruction I demande que l'on examine un nombre de segments qui peut être très élevé.

3° La même remarque que 2° est valable lorsque l'on veut déterminer si un opérande de I a été modifié.

Nous conclurons en disant que l'application textuelle de la définition D2 est à éviter.

La deuxième condition de la définition D2 est sans doute trop forte. En effet, dans l'ensemble $\mathcal{U}$ des chemins allant de S_o vers S, certains chemins ne sont que des chemins apparents en ce sens qu'ils ne seront peut-être jamais parcourus lors des exploitations du programme. Ainsi certaines instructions seront dites redondantes par l'application de la définition D2, alors que dynamiquement elles ne le sont pas. Mais la condition doit être imposée car, a priori, nous ne pouvons pas connaître statiquement le ou les chemins qui seront parcourus lors de l'exploitation du programme. Pour le problème que nous venons d'évoquer, il existe peut-être des solutions"hardware"(matérielles).

Le processus P_4 tel qu'il est défini en 2.2.2. pourrait s'énoncer de la façon suivante. Si le programme auquel on l'applique est constitué de l'ensemble $\mathcal{S} = \{ S_o, S_1, \ldots, S_n \}$ de segments :

Faire Examiner le segment S_i, déterminer les instructions redondantes dans ce segment, supprimer ces instructions et remplacer les références à ces instructions par les références adéquates.

Jusqu'à ce que la liste $\mathcal{S}$ soit épuisée.

Il est possible, par l'application de la division d'un graphe en niveaux étendus d'ordonner la liste de telle sorte que l'on minimise le temps pris par le processus P_4.

Le processus P_4 ainsi défini souffre des mêmes défauts que l'algorithme précédent puisqu'il fait appel à celui-ci. Nous signalons qu'il existe des algorithmes généraux permettant de déterminer si une instruction est redondante ou pas, mais ils sont essentiellement théoriques [COCKE 70].

2.2.4. Restriction des définitions D2, D3

Nous restreignons ici notre champ d'investigations aux segments d'instructions, nous définirons un processus P_4 restreint.

Définition D'2

Une instruction intermédiaire I appartenant à un segment S est redondante

si et seulement si

1° Il existe dans S une instruction J précédant I et formellement identique à I.

2° Entre l'occurrence de J et l'occurrence de I dans S, il n'existe pas d'instructions modifiant la valeur des opérandes de I.

Définition D'3

Le processus P'_4 (ou P_4 restreint) "Suppression d'instructions redondantes restreint aux segments d'instructions" est le processus qui supprime dans un segment d'instructions les instructions satisfaisant à la définition D'2 et qui remplace toute référence à l'instruction I de la définition D'2 par une référence à l'instruction J.

Exemples Soit le segment d'instructions S,

S : begin d:= b * c + d; a:= b * c + d; c:= b * c + d end

α_1(*; b; c)	α_1(*; b; c)
α_2(+; α_1; d)	α_2(+; α_1; d)
α_3(:=; d; α_2)	α_3(:=; d; α_2)
α_4(*; b; c)	α_4(+; α_1; d)
α_5(+; α_4; d)	α_5(:=; a; α_4)
α_6(:=; a; α_5)	α_6(:=; c; α_4)
α_7(*; b; c)	
α_8(+; α_7; d)	
α_9(:=; c; α_8)	
S en forme intermédiaire (a)	P'_4(S) (b)

Dans l'exemple ci-dessus, les instructions $\alpha_4(a)$ et $\alpha_7(a)$ sont redondantes et ont été éliminées dans (b); de plus, toute référence à ces instructions a été remplacée par une référence à l'instruction $\alpha_1(a)$.
Bien qu'elle soit formellement identique à l'instruction $\alpha_2(a)$, l'instruction $\alpha_5(a)$ n'est pas redondante car $\alpha_3(a)$ modifie la valeur de d. Cependant $\alpha_8(a)$ est redondante avec $\alpha_5(a)$.

Nous avons choisi des triples indirects comme forme intermédiaire, ce qui implique que les instructions formellement identiques ont déjà été détectées et que, de plus, le remplacement d'une référence à une instruction redondante ne doit plus être fait.
Le segment S se représente sous forme de triples indirects de la façon suivante :

TSQ	TSQ	T_I
α_1	α_1	α_1(*; b; c)
α_2	α_2	α_2(+; α_1; d)
α_3	α_3	α_3(:=; d; α_2)
α_1	α_2	α_4(:=; a; α_2)
α_2	α_4	α_5(:=; c; α_2)
α_4	α_5	
α_1		
α_2		
α_5		
S non optimisé	$P'_4(S)$	

Cette remarque nous conduira à la définition du processus P''_4 qui est l'équivalent du processus P'_4 pour des triples indirects.

2.2.5. Définition D''_3

Le processus P''_4 est le processus qui supprime, dans un segment d'instructions, les instructions satisfaisant à la définition D'_2.

Si grâce aux triples indirects, les instructions formellement identiques sont déjà déterminées, il nous reste un point crucial: vérifier la 2ème condition de la définition D'_2. Pour ce faire, nous étudierons dans le paragraphe suivant la dépendance entre les items de données utilisés ou définis dans un segment d'instructions.
La fonction de remplacement d'une référence à une instruction redondante par une référence à l'instruction qui a provoqué la redondance n'est plus nécessaire. Ceci est dû à la structure de la forme intermédiaire.

2.2.6. Analyse de la dépendance des items de données dans un segment d'instructions

L'ordre dans lequel se présentent les instructions intermédiaires dans un segment d'instructions est imposé par l'ordre dans lequel le programmeur a écrit les instructions sources. Cet ordre source est essentiellement linéaire mais n'est en réalité qu'un ordre fictif. Bien souvent, deux instructions consécutives dans un programme peuvent être permutées sans que l'on n'affecte le résultat du programme.

Exemple : a:= b + c; z:= e + f; est équivalent à la séquence z:= e + f; a:= b + c; par contre la séquence a:= b + c; z:= a * c; n'est pas équivalente à z:= a * c; a:= b + c.

D'une façon générale, nous pouvons énoncer l'assertion suivante.

Un programme P peut être considéré comme étant une suite d'instructions $I_1, I_2, \ldots, I_j, I_{j+1}, \ldots, I_n$ $(n < \infty)$.

Si I_j et I_{j+1} sont deux instructions telles qu'après leur exécution, les instructions suivantes qui seront exécutées sont respectivement I_{j+1}, I_{j+2}.

Alors Le programme $P' \equiv I_1, I_2, \ldots, I_{j+1}, I_j, I_{j+2}, \ldots, I_n$ est équivalent (*) au programme P

si et seulement si

1° I_j ne modifie la valeur d'aucun opérande de I_{j+1}

2° I_{j+1} ne modifie la valeur d'aucun opérande de I_j.

Cette assertion, que nous ne justiefierons pas, nous montre que l'ordre dans lequel le programmeur écrit ses instructions n'est pas toujours un ordre qui répond à des critères impératifs.

L'assertion montre qu'il existe un autre ordre qui, lui, doit être respecté si l'on désire conserver le caractère d'équivalence. Cet ordre est celui qui nous a permis d'affirmer que la séquence a:= b + c; z:= a * c; n'était pas équivalente à la séquence z:= a * c; a:= b + c.

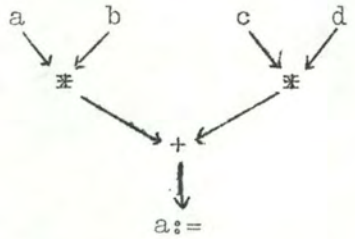
Comme le montre l'assertion, cet ordre dépend de la relation entre les items de données qui sont manipulés par les instructions du programme.

(*) Equivalent signifie que P et P' délivrent les mêmes résultats pour les mêmes conditions initiales et pour les mêmes données fournies aux deux programmes.

Généralement, on représente cet ordre par un arbre de dépendance (Dependency tree) dans lequel un sommet représente le résultat d'une opération (opération intermédiaire) et chaque arc entrant (Input link) un opérande pour l'opération. Des sommets n'ayant pas d'arcs entrants sont des références à des items de données (fetches of operands).

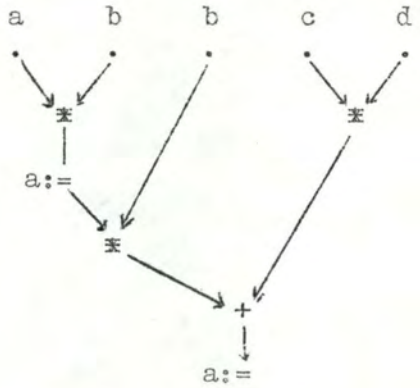
<u>Exemples</u> a:= b;

a:= a * b + c * d



a:= a * b

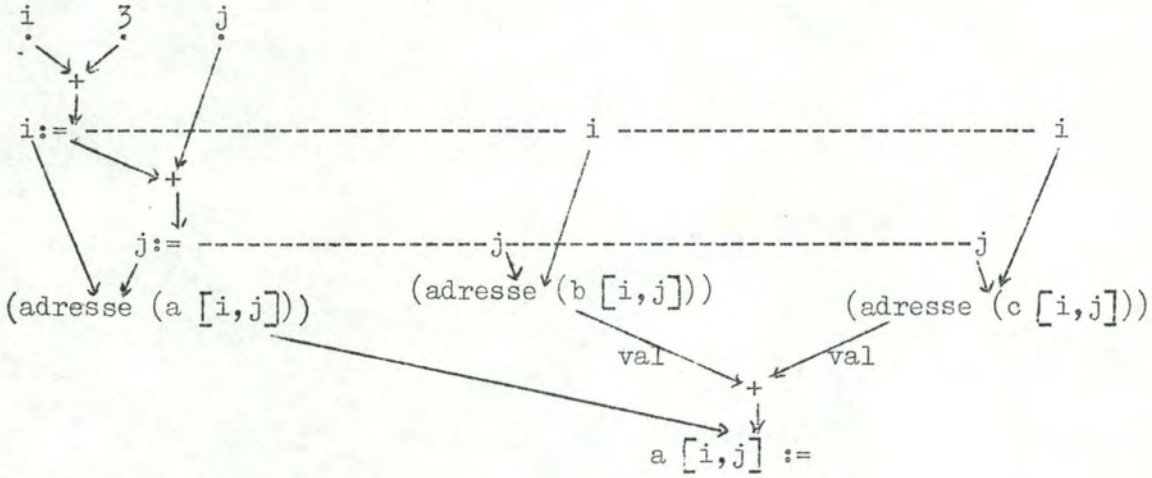
a:= a * b + c * d



<u>Exemple pour des variables indicées</u>

<u>array</u> a [1:25; 1:25] , b [1:25; 1:25] , c [1:25; 1:25]

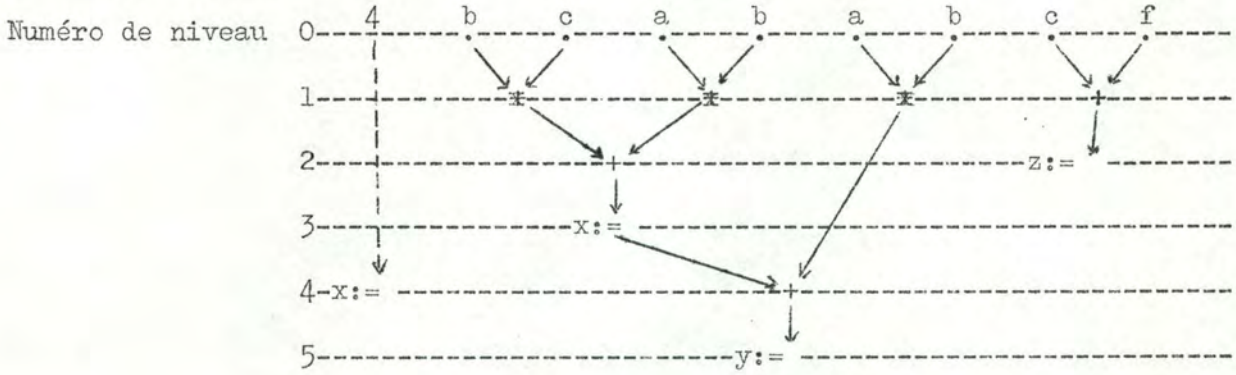
i:= i + 3; j:= j + i; a [i,j] := b [i,j] + c [i,j]



Il est à remarquer que, pour des tableaux, la référence à un de ces éléments dépend d'un calcul d'adresse dépendant des opérations qui ont été effectuées sur les indices.

L'arbre de dépendance n'est pas un arbre au sens de la théorie des graphes. Il est un graphe non nécessairement simplement connexe mais dont les composantes simplement connexes sont des arbres. Une dénomination plus correcte pour l'arbre de dépendance serait <u>forêt de dépendance</u>. Les composantes simplement connexes de la forêt de dépendance étant des graphes sans circuit, nous pouvons les décomposer en niveaux de telle façon que le niveau zéro soit attribué aux sommets n'ayant pas d'arcs entrants.

<u>Exemple</u> (1) <u>begin</u> x:= (b * c) + (a * b); y:= x +(a * b); z:= c + f <u>end</u>

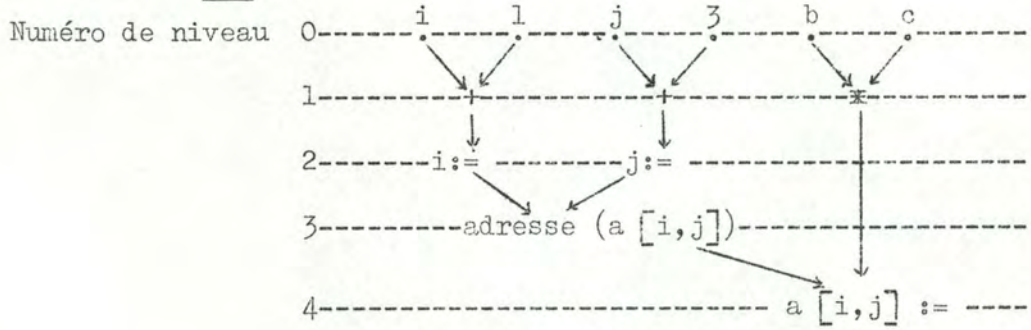


(a)

(2) <u>begin</u> <u>array</u> a [1:25; 1:25] ; --------

i:= i + 1; j:= j + 1; a [i,j] := b * c; ---

<u>end</u>



Remarque: Si nous avions la séquence

```
begin  x:= (b * c) + (a * b);
       y:= x + (a * b);
       z:= e + f;
       x:= 4  end   au lieu de la séquence (1),
```

l'opération x:= 4 devrait se voir attribuer un numéro de niveau égal à 4 au moins car elle ne pourrait être exécutée avant que l'on ait exécuté et employé le résultat de l'assignation x:= (b * c) + (a * b). (en pointillé sur la figure (a)).

Si nous faisons un parallèle avec les segments d'instructions, nous pouvons tirer les conclusions suivantes :

1° Une instruction intermédiaire n'aura jamais un numéro de niveau égal à zéro. Il n'y a que les références à des opérandes (items de données) qui peuvent posséder un numéro de niveau égal à zéro.

2° La première instruction d'un segment d'instructions aura toujours un numéro de niveau égal à 1, car cette instruction ne dépend des résultats d'aucune autre instruction la précédant dans le segment d'instructions.

3° Une instruction intermédiaire qui référencie une autre instruction intermédiaire qui la précède dans le segment d'instructions aura un numéro de niveau au moins égal à 2.

C'est la technique des numéros de niveau que nous appliquerons pour déterminer si la condition 2 de la définition D'2 est vérifiée ou non.

2.2.7. Procédure attribuant des numéros de niveau aux instructions d'un segment d'instructions

Cette procédure utilise une zone de TSQ, à savoir DEP-V-MOS, qui contiendra le numéro de niveau de l'instruction intermédiaire.
La procédure dispose aussi de deux tables :

1° STKLEVELNUM dont une entrée a le format suivant :

iden	levelnumdef	levelnumuse

iden : contient l'identificateur intermédiaire d'une variable simple ou d'un tableau.

levelnumdef : contient le numéro de niveau de la dernière instruction intermédiaire qui a modifié la valeur de la quantité. (definition)

levelnumuse : contient le numéro de niveau le plus élevé parmi les numéros de niveau des opérations intermédiaires qui ont référencé la quantité.(use)

2° STKLEVELNUM II dont une entrée a le format suivant :

iden	levelnum

iden : contient l'identificateur d'une instruction intermédiaire qui peut être référencée par une autre instruction intermédiaire.

levelnum : contient le numéro de niveau de l'instruction intermédiaire.

Nous ne discuterons pas de l'organisation de ces tables mais il est clair qu'elle doit permettre une mise à jour et une consultation rapides. Nous penserions par exemple à une organisation "random".

Nous pouvons très bien nous passer de la table STKLEVELNUM II car dans TSQ, le niveau d'une instruction intermédiaire est repris. Cependant, il nous semble moins efficace d'utiliser TSQ que de consulter STKLEVELNUM II, de plus cela nous compliquerait la tâche.

Nous pourrions ajouter une zone à chaque entrée dans TI en mettant dans cette zone le numéro de niveau actuel de l'instruction intermédiaire, mais attention, TI ne contient qu'un seul exemplaire de l'instruction.

Propriété - Levelnumdef contiendra toujours le numéro de niveau maximum parmi les numéros de niveau des instructions qui ont modifié la quantité.

Nous ne démontrerons pas cette propriété ici, mais nous la prendrons comme critère pour la réalisation de notre procédure. Le lecteur pourra facilement la déduire de la définition de levelnumdef que nous avons donnée et de la structure d'un segment d'instructions.

2.2.7.1. Détermination des numéros de niveau du résultat du calcul d'un opérande

Soit ARG un opérande d'une instruction intermédiaire I.

Nous noterons n(ARG) le numéro de niveau de l'opérande.

(1) Si ARG est une constante Alors n(ARG) égale 0;

(2) Si ARG est une variable simple Alors

Si ARG figure dans STKLEVELNUM Alors

n(ARG) = levelnumdef.STKLEVELNUM

Sinon

n(ARG) = 0

(3) Si ARG est une référence à une instruction intermédiaire

Alors n(ARG) = levelnum de STKLEVELNUM II

(4) Si ARG est une variable indicée TAB [SUBS]

Alors n(ARG) = max (n(TAB), n(SUBS))

Pour trouver n(TAB), on procède comme si TAB était une variable simple.

Pour trouver n(SUBS), on procède comme si SUBS était un opérande quelconque.

2.2.7.2. Détermination des numéros de niveau d'une instruction intermédiaire I

Notation n(I) dénote le numéro de niveau d'une instruction intermédiaire I.

(1) Si I est une instruction du type "Computation"

Alors n(I) = 1 + max (n(ARG 1), n(ARG 2))

découle directement de l'arbre de dépendance.

(2) Si I est une instruction du type "Assignation"

Alors n(I) = 1 + max (levelnumdef (ARG 1), levelnumuse (ARG 1), n(ARG 2)).

En effet, le numéro de niveau d'une assignation doit être supérieur au numéro de niveau de toute instruction précédente modifiant la valeur de ARG 1 et aussi supérieur au numéro de niveau de toute instruction précédente qui référencia ARG 1. De plus, n(I) ne peut être inférieur ou égal à n(ARG 2). (Remarque du paragraphe 2.2.6)

(3) Si I est une instruction du type "Indexed Assignation"

Alors n(I) = 1 + max (levelnumdef (TAB 1), levelnumuse (TAB 1)
n(SUBS 1), n(ARG 2))

En effet, la modification d"un élément d'un tableau ne peut être effectuée avant une utilisation de la valeur d'un élément du tableau dans une instruction qui précède I ou avant une instruction qui modifie la valeur d'un élément du tableau. De plus, il est naturel que n(I) dépende de n(SUBS 1) et de n(ARG 2). Voir l'arbre de dépendance.

(4) Si I est une instruction du type "APS" (appel de procédure standard)

Alors n(I) = 1 + n-ARG 2

Nous supposons que l'appel de procédure standard (sin, cos, ...) ne modifie pas la valeur de l'argument de la procédure standard.

(5) Si I est une instruction du type "AP" (appel de procédure)

Alors FORTRAN IV

n(I):= 1 + max {levelnumdef (paramètre 1), levelnumuse (paramètre 1), ..., levelnumdef (paramètre n),
levelnumuse (paramètre n),
levelnumdef (opérande global 1),
levelnumuse (opérande global 1),
⋮
levelnumdef (opérande global m),
levelnumuse (opérande global m)}

Algol 60

n(I):= un numéro de niveau supérieur à tous les numéros de niveau actuellement employés.

Nous pourrions développer ici la même analyse que pour la propagation de constantes, cependant nous ne nous en tiendrons qu'à des conclusions. Pour rappel, nous signalons que nous avons supposé que seules des variables simples, temporaires ou indicées, de même que des tableaux pouvaient figurer dans la liste des paramètres actuels d'une procédure.

a) FORTRAN IV

Nous considérons que toute variable simple ou indicée, tout tableau figurant dans la liste des paramètres actuels, de même que ceux qui figurent

dans la liste d'un ordre COMMON sont modifiés par l'exécution de l'appel de procédure.

Par conséquent, l'appel de procédure peut se traiter de la même façon qu'une assignation multiple au point de vue de l'attribution d'un numéro de niveau.

S'il nous était possible de répondre à la question 3 dans le paragraphe 2.1.7.7., l'appel de procédure se comporterait au point de vue de l'attribution d'un numéro de niveau pour certains paramètres ou opérandes globaux, comme une assignation et pour d'autres, comme une instruction du type "computation".

b) Algol 60

Nous avons dit qu'un appel de procédure Algol 60 termine un segment d'instruction. Comme un appel de procédure se comporte généralement de la même façon qu'une assignation multiple, il faut s'arranger pour que le numéro de niveau soit supérieur au numéro de niveau de toute autre instruction du segment d'instructions.

(6) Si I est une instruction du type "Branchement"

Alors n(I) = un numéro de niveau supérieur à tous les numéros de niveau actuels.

En effet, une instruction de branchement doit rester la dernière instruction à exécuter d'un segment.

Nous connaissons maintenant les règles pour déterminer les numéros de niveau d'une instruction intermédiaire et d'un opérande d'une instruction.

2.2.7.3. Procédure d'attribution des numéros de niveau aux instructions d'un segment S

Un segment d'instructions S peut être considéré comme une suite I_1, I_2, ..., I_n d'instructions intermédiaires.
D'après les conclusions du paragraphe 2.2.6, I_1 a toujours un numéro de niveau égal à 1; "Trouver $n(I_i)$" lui attribuera ce numéro de niveau.

(1) i:= 1;

(2) Prendre I_i dans la liste S;

```
(3) Si  I_i est une instruction du type "ASSIGNATION"
    Alors Début
          Trouver n(I_i); Dep-V-Mos (I_i):
          Levelnumdef (ARG 1):= n(I_i);
          Si ARG 2 est une variable simple
          Alors  Début  Si Levelnumuse (ARG 2) < n(I_i)
                        Alors Levelnumuse (ARG 2):= n(I_i)
                 Fin;
    Si ARG 2 est une variable indicée (TAB 2 [SUBS 2])
    Alors  Début
           Si Levelnumuse (TAB 2) < n(I_i)
           Alors Levelnumuse (TAB 2):= n(I_i);
           Si SUBS 2 est une variable simple
           Alors  Début
                  Si Levelnumuse (SUBS 2) < n(I_i)
                  Alors Levelnumuse (SUBS 2):= n(I_i)
                  Fin
           Fin
    i:= i + 1;
    Si la liste S est épuisée  Alors terminer l'algorithme
                               Sinon continuer en (2)
    Fin;
(4) Si I_i est une instruction du type "Computation"
        (OP; ARG 1; ARG 2)
    Alors  Début
           Trouver n(I_i); Dep-V-Mos (I_i):= n(I_i);
           Levelnum (I_i):= (I_i);
           Si ARG 1 est une variable simple
           Alors  Début
                  Si Levelnumuse (ARG 1) < n(I_i)
                  Alors Levelnumuse (ARG 1):= n(I_i)
                  Fin;
```

```
Si ARG 1 est une variable indicée (TAB 1 [SUBS 1])
Alors Début
      Si  Levelnumuse (TAB 1) < n(I_i)
      Alors Levelnumuse (TAB 1):= n(I_i);
      Si SUBS 1 est une variable simple
      Alors Début
            Si Levelnum (SUBS 1) < n(I_i)
            Alors Levelnum (SUBS 1):= n(I_i)
            Fin;
Si ARG 2 est une variable simple
Alors idem que pour ARG 1 variable simple;
Si ARG 2 est une variable indicée
Alors idem que pour ARG 1 est une variable indicée;
i:= i + 1;
Si la liste S est épuisée Alors terminer l'algorithme
                          Sinon continuer en (2)
Fin (4);
(5) Si  I_i est une instruction du type "INDEXED ASSIGNATION"
        (:=; TAB 1 [SUBS 1]); ARG 2)
Alors Début
      Trouver n(I_i); Dep-V-Mos (I_i):= n(I_i);
      Levelnumdef (TAB 1):= n(I_i)
      Si SUBS 1 est une variable simple
      Alors Début
            Si Levelnumuse (SUBS 1) < n(I_i)
            Alors Levelnumuse (SUBS 1):= n(I_i)
            Fin
Si ARG 2 est une variable simple
Alors Début
      Si Levelnumuse (ARG 2) < n(I_i)
      Alors Levelnumuse (ARG 2):= n(I_i)
      Fin;
```

```
Si ARG 2 est une variable indicée
Alors Début
      Si Levelnumuse (TAB 2) < n(I_i)
      Alors Levelnumuse (TAB 2):= n(I_i);
      Si SUBS 2 est une variable simple
      Alors Début
            Si Levelnumuse (SUBS 2) < n(I_i)
            Alors Levelnumuse (SUBS 2):= n(I_i)
            Fin;
      Fin;
i:= i + 1;
Si la liste S est épuisée Alors terminer l'algorithme
                          Sinon continuer en (2)
Fin (5);
```

```
(6) Si I_i est une instruction du type "APS" (APS; P.S; ARG 2)
    Alors Début
          Trouver n(I_i); Dep-V-Mos (I_i):= n(I_i)
          Levelnum (I_i):= n(I_i)
          Si ARG 2 est une variable simple
          Alors Début
                Si Levelnumuse (ARG 2) < n(I_i)
                Alors Levelnumuse (ARG 2):= n(I_i)
                Fin;
          Si ARG 2 est une variable indicée
          Alors Début
                Si Levelnumuse (TAB 2) < n(I_i)
                Alors Levelnumuse (TAB 2):= n(I_i)
                Si SUBS 2 est une variable simple
                Alors Début
                      Si Levelnumuse (SUBS 2) < n(I_i)
                      Alors Levelnumuse (SUBS 2):= n(I_i)
                      Fin
                Fin;
    i:= i + 1;
    Si la liste S est épuisée Alors terminer l'algorithme
                              Sinon poursuivre en (2)
    Fin (6); *
```

* Nous avons implicitement supposé qu'un APS ne modifie pas la valeur de ARG2.

(7) Si I_i est une instruction de branchement
Alors Début Trouver $n(I_i)$;
Dep-V-nos $(I_i) := n(I_i)$;
Terminer l'algorithme
Fin (7);

(8) Si I_i est une instruction du type "AP"
Alors Début
(a) Fortran IV
Trouver $n(I_i)$; Dep-V-nos $(I_i) := n(I_i)$;
Si la procédure est une procédure fonction
Alors Levelnum $(I_i) := n(I_i)$;
Comment Une procédure se comporte également comme une utilisation de résultats antérieurement obtenus du point de vue de la modification des numéros de niveau d'utilisation "Level numuse". Dès lors il faut modifier les zones Levelnumuse de tous les paramètres actuels et tous les opérandes globaux.
$i := i + 1$;
Si la liste S est épuisée Alors terminer l'algorithme
Sinon poursuivre en (2);

(b) Algol 60
Trouver $n(I_i)$; Dep-V-nos $(I_i) := n(I_i)$;
Terminer l'algorithme;
Fin (8);

2.2.7.4. Critique de l'algorithme

1) Nous aurions très bien pu nous passer de la zone levelnumuse dans STKLEVELNUM. En effet, la règle de recherche du numéro de niveau d'une instruction modifiant la valeur d'un opérande impose que ce numéro de niveau soit supérieur aux numéros de niveau de toutes les instructions précédentes qui ont modifié la valeur de cet opérande et aussi supérieur aux numéros de niveau de toutes les instructions précédentes qui ont référencé l'opérande.

Si nous prenons la règle suivante: le numéro de niveau d'une variable A ou d'un tableau T devient égal au numéro de séquence dans le segment d'instructions, du triple indirect modifiant la valeur de l'opérande A ou T, nous arrivons à des résultats semblables.

C'est la règle utilisée par David GRIES dans son ouvrage "Compiler Construction for digital Computers". Il faut pourtant remarquer qu'avec cette règle, nous ne reproduisons pas exactement la structure d'arbre de dépendance. Cette règle facilite certainement l'implémentation mais nous avons choisi l'autre pour des motifs de lisibilité et de compréhension.

2) Pour l'attribution des numéros de niveau à des variables indicées, nous aurions pu être plus raffinés. En effet, dans notre système, une fois qu'une instruction modifie la valeur d'un élément d'un tableau, nous supposons que tous les éléments du tableau ont été modifiés. Nous pourrions tester si l'élément modifié est un élément exactement localisable, c'est-à-dire si les indices de l'élément ont une valeur actuellement connue. Ceci compliquerait fortement l'algorithme et sans doute pour n'améliorer que très peu le programme.

2.2.8. Algorithme de "suppression des instructions redondantes"

Les numéros de niveau seront utilisés de la façon suivante.
Si dans un segment d'instructions $S = I_1, I_2, \ldots, I_i, \ldots, I_j, \ldots, I_n$, l'instruction I_i est formellement identique à I_j et que $n(I_i) = n(I_j)$, alors I_j est redondante.

Nous démontrerons l'assertion pour un type d'instructions. La démonstration est analogue pour les autres types d'instructions.

Démonstrations

(1) - Soit $I_i \equiv I_j \equiv$ (OP; ARG 1; ARG 2) où OP est quelconque et ARG 1 et ARG 2 sont deux variables simples.

(2) - Par définition de n(I); I étant une instruction, nous avons

$$n(I_j) = n(I_i) > n(\text{ARG 2}) = \begin{cases} 0 \text{ ou} \\ \text{levelnumdef (ARG 2)} \end{cases}$$

$$n(I_j) = n(I_i) > n(\text{ARG 1}) = \begin{cases} 0 \text{ ou} \\ \text{levelnumdef (ARG 1)} \end{cases}$$

(3) Supposons que I_j ne soit pas redondante.

$\Rightarrow$ Il existe au moins une instruction I_k située entre I_i et I_j qui modifie un des opérandes de I_i ou de I_j. Soit ARG 1 est opérande et soit $I_k \equiv$ (:=; ARG 1; ARGK).

(4) $n(I_k) = 1 + \max$ (levelnumdef (ARG 1), levelnumuse (ARG 1), n(ARGK)) $>$ levelnumdef (ARG 1) (ancienne valeur)

En attribuant ce numéro de niveau à I_k, la procédure modifie également levelnumdef de ARG 1 et le rend égal à $n(I_k) >$ (levelnumdef (ARG 1) ancien).

(5) $n(I_j) = 1 + \max$ (levelnumdef (ARG 1); levelnumdef (ARG 2) ou zéro)

$> 1 + \max$ (levelnumdef (ARG 1) ou zéro; (ancien) levelnumdef (ARG 2) ou zéro)

$> n(I_i)$

ce qui est contraire aux hypothèses.

Il est à remarquer que nous avons ainsi jeté les bases de la démonstration de la condition suffisante pour qu'une instruction I soit redondante. Il est également possible de démontrer la condition nécessaire qui, de notre point de vue, offre moins d'intérêt.

Pour limiter le temps pris pour chercher des instructions I qui sont formellement identiques et qui ont même numéro de niveau, l'algorithme de suppression des instructions redondantes trie le segment S sur des numéros de niveau croissant. Après cela, il élimine toutes les instructions I qui sont redondantes; pour ce faire, il analyse un groupe d'instructions ayant même numéro de niveau et y supprime toutes celles qui sont formellement identiques sauf une.

2.2.9. Discussion sur "la suppression d'instructions redondantes"

2.2.9.1. Nous n'avons envisagé ici que la suppression d'instructions redondantes dans un segment d'instructions. Remarquons qu'il n'est pas nécessaire que 2 instructions I_i et I_j se trouvent initialement dans un même segment d'instructions pour que l'une soit dite "redondante". Mais par suite de déplacement d'instructions, les instructions I_i et I_j peuvent finir par se trouver dans le même segment.

Toutefois, nous n'affirmerons pas non plus que la combinaison de la technique de déplacement d'instructions avec P_4'' peut donnée le processus P_4.

2.2.9.2. De toute façon, il faut tenir compte dans une élimination globale d'instructions redondantes, des contraintes matérielles. En effet, l'élimination de sous-expressions communes (instructions redondantes) d'une façon globale nécessite presque toujours l'introduction de zones temporaires et donc l'introduction dans le code objet d'instructions de chargement et de déchargement, ce qui risque de rendre le code généré bien plus mauvais.
Cette remarque (2) n'est valable que pour des machines qui ne disposent pas d'un nombre suffisant de registres généraux et flottants, capables de mémoriser un résultat intermédiaire pendant une période de temps suffisamment longue. (Exemple IBM 360, SIEMENS 4004).
Le compilateur FORTRAN 4 sur IBM 360/OS réalise une élimination d'instructions redondantes globalement mais néanmoins très localement car le compilateur dispose d'un nombre trop faible de registres. Une analyse machine dépendante serait bienvenue pour résoudre des problèmes de ce type.

2.2.9.3. La suppression d'instructions redondantes serait plus efficace si nous étions à même d'utiliser la propriété de commutativité de certains opérateurs, à savoir + et *.

Exemple: A * B est équivalent à B * A, ou A + B + C est équivalent à C + B + A.

Nous pouvons reprendre ici une proposition de David GRIES. Il s'agit d'ordonner les opérandes dans un certain ordre. Soit une expression qui ne comporte pour opérateur que "+", nous plaçons en tête de l'expression tous les termes qui ne sont pas des constantes ou des variables, puis toutes les variables indicées en ordre lexical, puis toutes les variables simples en ordre lexical, ensuite toutes les constantes.

<u>Exemple</u>

A:= B + 1 + C + 2; B:= 6 + B + C devient A:= B + C + 1 + 2; B:= B + C + 6
ce qui nous permettra de ne calculer qu'une seule fois B + C et de transformer 1 + 2 en 3.

La même règle d'ordonnancement des opérandes est valable pour l'opérateur "*". Lorsque des expressions combinent à la fois + et *, le problème se complique mais nous proposerions la solution suivante.

(1) Isoler toutes les séquences (Termes) comprises entre deux "+"

(2) Ordonner les séquences qui ne comportent que des "*" comme opérateurs, suivant l'ordre proposé par Gries.

(3) Ordonner lexicalement l'ensemble des termes de l'expression.

Exemple

x:= b * a + c * 2 * d * 3; (a)

y:= a * b + c * d * e * f + d * c * 2 * 3 (b)

devient si (3) donne l'ordre suivant des termes de (a)

ba < cd23 et des termes de (b) ab < cd23 < cdef

x:= a * b + c * d * 2 * 3;

y:= a * b + c * d * 2 * 3 + c * d * e * f

ce qui est équivalent à la séquence optimisée suivante

T = c * d; x:= a * b + T * 6; y:= x + T * e * f

La règle que nous proposons n'est pas valable si l'expression comporte des parenthèses. Pour ces cas, il vaut mieux parfois enlever les parenthèses et appliquer ensuite la règle, parfois il vaut mieux garder les parenthèses.

A * (E + F) * B = A * E * B + A * F * B

= A * B * E + A * B * F

Malgré tout, ceci ne nous permet pas de résoudre le problème suivant.

Soit l'expression A:= B + C + C + B. En adoptant la règle de GRIES, on obtiendrait A:= B + B + C + C, alors que B + C ne devrait être calculé qu'une seule fois.

Pour certaines machines, C et B étant des entiers, 2 * (B + C) serait une forme optimale car on pourrait utiliser un shift à gauche. Pour d'autres machines, ce ne serait pas le cas.

Le problème soulevé ici est un problème qui n'a pas reçu actuellement, semble-t-il, de solution satisfaisante malgré l'abondante littérature traitant ce sujet. Nous signalons ici à titre documentaire les articles de

- Knuth, D.E., "Runcible - algebraic translation on a limited computer", CACM 2 (janvier 1959) (18-21).

- Huskey, H.D., "Compiling techniques for algebraic expressions", Computer Journal 4 (61) (10-19).

- Hopgood, F.R.A., "Compiling Techniques", American Elsevier, New York, 1969.

- Floyd's, "An algorithm for coding efficient arithmetic operations", CACM 4 (janvier 1961)(42-51).

- Brueuer, "Generation of optimal code for expressions via factorization", CACM 12 (juin 1969)(333-340).

- Fateman, "Optimal code for serial and parallel computation", CACM 12 (décembre 1969)(694-695).

2.2.9.4. Dans l'algorithme que nous avons présenté, certaines opérations intermédiaires figurant dans un segment d'instructions seront dites non redondantes alors qu'elles le sont effectivement.

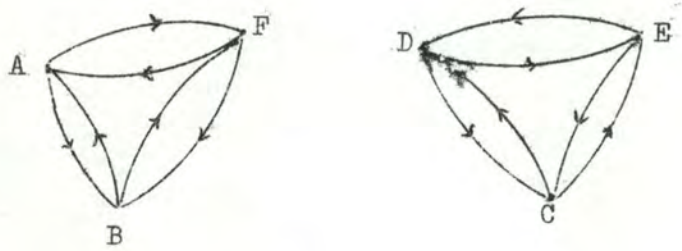
<u>Exemple</u>

(1) (:=; A; B)
(2) (:=; C; B)
⋮ } ⟶ pas de modifications de A, B ou C.
(i) (*; A; D)
(j+1) (*; C; D)
(j+2) (*; B; D)

Les triples j+1, j+2 sont redondants vu que A = B = C. Notre algorithme ne nous permet cependant pas de les détecter. Pour résoudre ce problème, COCKE J. et SCHWARTZ, J.T. ont proposé un algorithme se basant sur la notion de "value number". Il serait trop long d'exposer ici la démarche de cet algorithme qui est d'une conception fortement différente de la nôtre et surtout beaucoup plus sophistiqué. [COCKE 70]

Par contre, nous pensons que le problème est résoluble au niveau des routines sémantiques du compilateur qui produisent le code intermédiaire. Nous donnerons ici quelques idées de base d'un algorithme possible. Nous nous aiderons d'un exemple.

Soit la séquence A:= B; F:= B; D:= E; C:= E.
Vu que l'assignation implique l'égalité, c'est-à-dire que A:= B implique que A et B deviennent égaux, nous pouvons représenter cette séquence de la manière suivante, au moyen d'un graphe dans lequel un arc symbolise la relation d'égalité entre deux variables.



Les arcs (A,F), (F,A), (D,E) et (E,D) sont rajoutés au graphe grâce à la propriété de transitivité de la relation d'égalité.
Ou, sous forme matricielle

	A	B	C	D	E	F
A	1	1	0	0	0	1
B	1	1	0	0	0	1
C	0	0	1	1	1	0
D	0	0	1	1	1	0
E	0	0	1	1	1	0
F	1	1	0	0	0	1

Pour chaque composante fortement connexe de ce graphe (la propriété de forte connexité découle des propriétés de l'égalité), nous prenons un sommet type représentant la composante fortement connexe (graphe réduit). Chaque référence à une variable appartenant à une même composante fortement connexe que le sommet type, nous la remplaçons par une référence au sommet type.

<u>Exemple</u> A:= B; F:= B; D:= E; C:= E;
z:= A + D; y:= z * F + z * C;
devient
A:= B; F:= B; D:= E; C:= E
z:= B + E; y:= z * B + z * B;

Nous arrêterons ici nos considérations en signalant que plusieurs problèmes de cet ordre restent à résoudre. Toutefois, l'intérêt de ces problème n'est pas mineur car nous voyons que quatre variables A, F, D, C peuvent peut-être disparaître du programme.

2.2.9.5. La suppression des instructions redondantes est-elle nécessaire ? Et ne vaudrait-il pas mieux former les programmeurs ?

Nous appuierons notre discussion sur deux exemples, l'un extrait de Allen et l'autre de Schwartz et Cocke.

On peut dire que les instructions redondantes apparaissent dans des langages évolués pour deux raisons :

1) Garantir une plus grande lisibilité des programmes. Par exemple, il est plus naturel d'écrire le programme qui cherche les racines d'une équation du second degré de la façon suivante :

```
x:= (-b + sqrt (b ** 2 - 4.* a * c))/(2.* a);
y:= (-b - sqrt (b ** 2 - 4.* a * c))/(2.* a);
```

que de la façon suivante :

```
d:= 2.* a;
f:= sqrt (b ** 2 - 4.* a * c);
x:= (-b + f)/d;
y/= (-b - f)/d;
```

2) Même si le programmeur a été très attentif pour éviter d'écrire des instructions redondantes, il en existe qu'il ne peut pas éviter.

Exemple

```
begin real array a [1:25; 1:25] , b[1:25, 1:25] ;
      integer i, j;
        ↓
        a [i,j] := b [i,j] ;
        ↓
end
```

sera traduit par un compilateur non optimisant:

```
T 1 := i * 25;
T 2 := T 1 + j;
T 3 := adresse (b) + T 2;
Charger à partir de l'adresse (T 3)
T 4 := i * 25;
T 5 := T 4 + j;
T 6 := adresse (a) + T 5;
Décharger dans l'adresse (T 6)
```

Il est évident que les quantités T 1 et T 4 ont même valeur, de même que T 2 et T 5. Par conséquent, par la suppression d'instructions redondantes, nous

arriverions à la séquence suivante :

T 1 := i * 25;

T 2 := T 1 + j;

T 3 := adresse (b) + T 2;

Charger à partir de l'adresse (T 3);

T 4 := adresse (a) + T 2;

Décharger dans l'adresse (T 6);

Pour répondre à la question "Ne vaudrait-il pas mieux former les programmeurs ?". Nous dirions que les points 2.2.9.3, 2.2.9.4 et bien d'autres encore pourraient être enseignés aux programmeurs, mais qu'un programmeur trop raffiné écrit des programmes à peu près illisibles.
Nous pouvons citer comme exemple les programmes écrits par H. Rutishauser dans son ouvrage "Description of Algol 60": ils sont sans doute fort optimaux mais très peu lisibles.

Chapitre III

TOPOLOGIE DES PROGRAMMES ET DEPENDANCE DES ITEMS DE DONNEES DANS L'ENSEMBLE D'UN PROGRAMME

Dans ce chapitre, nous envisageons les concepts nécessaires à une optimisation globale. Ces concepts sont ceux de la théorie des graphes, adaptés aux besoins de la cause.

3.1. Décomposition d'un Graphe en Régions.

3.2. Quelques concepts supplémentaires.

3.3. Dépendance des items de données dans l'ensemble d'un programme.

La plupart des idées de ce chapitre sont extraites de l'ouvrage de David Gries [GRIES 71], également de l'article de F.E. Allen [ALLEN 69]. D'autres idées sont originales ou extraites d'articles dont les références sont données au moment adéquat.
Pour des informations complémentaires sur la théorie des graphes, nous renvoyons le lecteur au cours de "Théorie des graphes" de Monsieur Fichefet [FICHEFET 73].

3.1. Décomposition d'un graphe en régions

Différents concepts de la théorie des graphes sont intéressants pour l'optimisation que nous envisageons dans les chapitres suivants. Il arrive cependant souvent qu'ils soient trop faibles ou trop forts que pour permettre une optimisation sophistiquée qui rendrait un programme écrit en langage évolué aussi efficient que ce même programme écrit en langage de base par un programmeur idéal. De ce fait, les chercheurs sont souvent amenés à les adapter ou à en introduire de nouveaux. Nous nous aiderons dans ce paragraphe des concepts classiques et de certains concepts que nous avons adaptés aux besoins de la cause.

La plupart des idées de ce paragraphe sont extraites de l'ouvrage de David Gries [GRIES 71], également de l'article de F.E. Allen [ALLEN 69]. D'autres idées sont originales ou extraites d'articles dont les références seront données au moment adéquat.

3.1.1. Rappels

Nous rappellrons ici les définitions de certains concepts que nous utiliserons dans la suite de ce travail.

Successeur

Un sommet x_i d'un graphe G est un "successeur" d'un sommet x_j s'il existe dans G un chemin allant de x_j à x_i. x_i est un successeur immédiat de x_j si le chemin est de longueur 1 (1 arc).

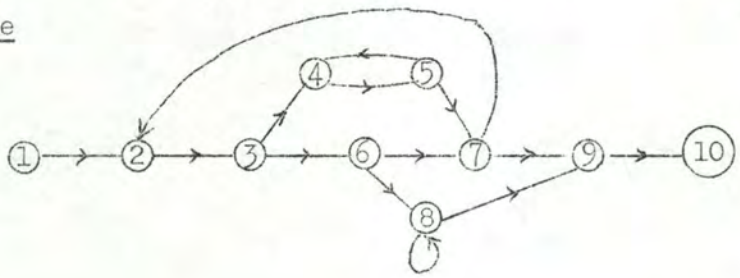
Prédécesseur

Un sommet x_i est un "prédécesseur" d'un sommet x_j s'il existe dans un chemin allant de x_i à x_j.
x_i est un prédécesseur immédiat de x_j si le chemin est de longueur 1 (1 arc).

Région fortement connexe d'un graphe

Une région fortement connexe d'un graphe G est un sous-graphe SG de G tel que, quels que soient deux sommets x_i et x_j de SG, il existe un chemin dans SG allant de x_i vers x_j et un chemin allant de x_j vers x_i.

Exemple



Les régions fortement connexes de ce graphe sont :

$\{8\}$, $\{4,5\}$, $\{2, 3, 6, 7\}$, $\{2, 3, 4, 5, 6, 7\}$, $\{2, 3, 4, 5, 7\}$.

Dorénavant, nous dirons Région pour région fortement connexe.

Il faut noter que la notion de région fortement connexe n'est pas identique à la notion de composante fortement connexe d'un graphe.
Une composante fortement connexe d'un graphe est une région fortement connexe telle qu'il n'existe aucune autre région fortement connexe la contenant strictement. Il en résulte que les composantes fortement connexes d'un graphe forment une partition de ce graphe alors que les régions fortement connexes constituent un recouvrement de ce graphe.
On donnera plus loin un algorithme permettant de décomposer un graphe en régions fortement connexes.

On comprendra aisément par la suite les raisons qui nous ont poussé à utiliser des régions fortement connexes plutôt que des composantes fortement connexes du graphe du programme.

Sommet d'entrée d'une région

Un sommet d'entrée d'une région R est un sommet x_i de R tel qu'il existe un sommet x_j n'appartenant pas à R d'un arc allant de x_j vers x_i.

Exemple : Pour la région $\{8\}$, 8 est un sommet d'entrée de même que 4 pour $\{4,5\}$, 7 et 2 pour $\{2, 3, 6, 7\}$ et 2 pour $\{2, 3, 4, 5, 6, 7\}$.
Dorénavant, nous parlerons d'entrée d'une région ou encore de point d'entrée d'une région.

Sommet de sortie d'une région

Un sommet de sortie d'une région R est un sommet x_i de R tel qu'il existe un sommet s_j n'appartenant pas à R et un arc allant de x_i vers x_j.

Exemple : Pour la région $\{8\}$, 8 est un sommet de sortie, de même que 5 pour $\{4, 5\}$, 6 et 7 pour $\{2, 3, 4, 5, 6, 7\}$, 6, 7 et 3 pour $\{2, 3, 7, 6\}$.
Dorénavant, nous dirons sortie d'une région.

Sommet prédécesseur d'une région

Un sommet x_i est un prédécesseur d'une région R s'il est un prédécesseur d'un sommet d'entrée de R. Il est un prédécesseur immédiat s'il est un prédécesseur immédiat d'une entrée de R.

Sommet successeur d'une région

Un sommet x_j est un successeur d'une région R s'il est un successeur d'une sortie de R. Il est un successeur immédiat s'il est un successeur immédiat d'une sortie de R.

3.1.2. Régions et liste de régions

Avant l'optimisation, le programme divisé en segments d'instructions et mis sous forme de graphe est analysé en vue de construire une liste R de régions $R_1, R_2, \ldots, R_n$ telle que :

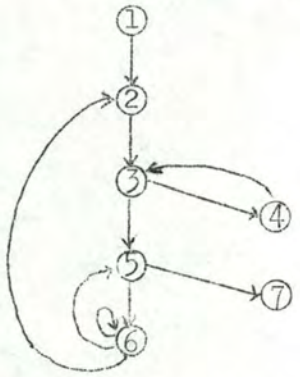
1) $\forall\, i,j \in \{1,\ldots, n\} \quad R_i \neq R_j$

2) $\forall\, i,j \in \{1,\ldots, n\}$ et $i < j$

$(R_i \cap R_j = \emptyset)$ ou $(R_i \cap R_j = R_i)$

(ou exclusif)

Exemple



Liste de régions R pour le graphe

$\{6\}$, $\{5, 6\}$, $\{2, 3, 4, 5, 6\}$

Remarquons que $\{3, 4\}$ et $\{2, 3, 5, 6\}$ sont des régions mais ne peuvent figurer dans la liste R car la condition 2 serait violée. La réunion de ces deux régions doit cependant se trouver dans la liste R.

En ce qui concerne les sommets 1 et 7, ils ne figurent dans aucune région. Ils forment en quelque sorte une liste de sommets isolés. Les instructions figurant dans ces segments ne seront exécutées qu'une seule fois.

3.1.3. Utilisation de la liste R par l'optimiseur

L'optimiseur utilise la liste des régions $R = R_1, R_2, \ldots, R_n$ de la façon suivante :

1) Les régions sont séquentiellement prises dans la liste R pour être traitées par l'optimiseur. Par conséquent, lorsque la région R_i est optimisée, les régions $R_1, R_2, \ldots, R_{i-1}$ ont déjà été traitées.

2) Si la région R_i, que l'on désire traiter, couvre certaines régions déjà examinées, seuls les segments non encore traités dans R_i seront optimisés. Pour dét rminer quelles sont les quantités qui sont modi iées ou référencées dans la région R_i, l'ensemble des segments de R_i est utilisé.

L'ensemble des segments non encore optimisés dans R_i est donné par la formule suivante

$$R'_i = R_i \cap \left(\bigcap_{j=1}^{i-1} \overline{R}_j \right)$$

où $\overline{R}_j = G \setminus R_j$

3) Lorsque, suite à l'optimisation d'une région R_i, certaines instructions invariantes dans R_i sont déplacées, celles-ci

1° seront insérées dans des segments prédécesseurs ou successeurs immédiats de R_i;

2° soit formeront de nouveaux segments qui seront placés sur les chemins entrant dans R_i ou sur les chemins sortant de R_i.

4) Lorsque de nouveaux segments sont générés suite à l'optimisation d'une région R_i, les entrées R_{i+1}, R_{i+2}, ..., R_n de la liste R seront modifiées en conséquence.

5) Les segments générés suite à l'optimisation de R_i ne sont pas rendus discriminables par rapport aux segments originaux. Ceci a la conséquence suivante: lorsque des régions qui se couvrent sont optimisées, des instructions peuvent être déplacées plusieurs fois, s'écartant ainsi de leur position initiale dans le programme.

<u>Exemple</u> : Reprenons l'exemple du paragraphe 3.1.2.
Si l'optimisation de la région $\{6\}$ génère un nouveau segment, soit ⑧, qui doit être inséré entre les segments 5 et 6, la liste R sera modifiée de la façon suivante : $\{6\}$, $\{5, 8, 6\}$, $\{2, 3, 4, 5, 8, 6\}$.
La région suivante à optimiser est $\{5, 8, 6\}$; dans celle-ci, seuls les segments 5, 8 seront retenus.
Supposons que l'optimisation génère un nouveau segment ⑨ à insérer entre les segments 3 et 5; la liste R se présentera dès lors comme suit :
$\{6\}$ $\{5, 6, 8\}$, $\{2, 3, 4, 5, 6, 8, 9\}$.
Finalement, les segments ②, ③, ④, ⑨ sont examinés et les instructions

déplacées de ces segments sont insérées dans le segment 1. Ainsi, des instructions figurant dans le segment 6 initialement peuvent figurer, en fin d'optimisation dans le segment 1.

3.1.4. Critique de la décomposition d'un programme en régions

a) Avantages de la liste de régions

1) L'optimisation d'un programme peut se faire de façon modulaire. Si l'on ne désire pas optimiser l'entièreté d'un programme, on peut arrêter l'optimisation après le traitement d'une région quelconque. De plus, si l'on dispose d'informations relatives à la fréquence d'exécution de certaines régions, on peut demander à l'optimiseur de ne traiter que les régions qui ont une fréquence élevée et même, dans ces régions, ne sélectionner, à des fins d'optimisation, que des segments bien déterminés.

2) Les techniques d'optimisation développées pour une optimisation orientée régions ne demandent aucune information relative à l'environnement de la région, c'est-à-dire relative aux segments qui ne figurent pas dans la région et à la façon dont ces segments sont reliés entre eux. Exemple : il n'est pas nécessaire de savoir que la variable A est modifiée en dehors de la région R_i pour déterminer si l'opération OP; A; B est invariante dans R_i.

b) Problèmes relatifs à la liste de régions

Reprenons l'exemple du paragraphe 3.1.2. La décision de ne pas faire figurer les régions $\{3, 4\}$ et $\{2, 3, 5, 6\}$ dans la liste R est quelque peu arbitraire. En effet, les deux listes de régions suivantes sont également correctes :

R_1 : $\{6\}$, $\{5, 6\}$, $\{3, 4\}$, $\{2, 3, 4, 5, 6\}$

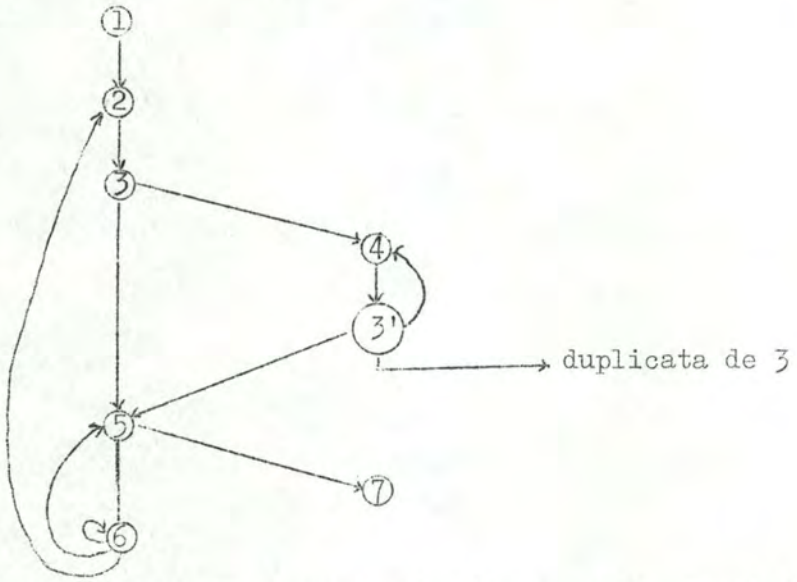
R_2 : $\{6\}$, $\{5, 6\}$, $\{2, 3, 5, 6\}$, $\{2, 3, 4, 5, 6\}$

Comme $\{3, 4\}$ et $\{2, 3, 5, 6\}$ ne peuvent pas figurer en même temps dans la liste R et que les listes R_1 et R_2 sont des listes correctes, quel choix faut-il faire ?

1) Pour résoudre ce problème de choix, nous pourrions proposer de supprimer $\{3, 4\}$ et $\{2, 3, 5, 6\}$ de la liste R mais cette solution est la moins bonne. En effet, supposons qu'il existe dans la boucle

$\{3, 4\}$ une opération (OP; A; B) telle que A et B ne dépendent d'aucun calcul exécuté dans la boucle $\{3, 4\}$; dès lors, l'opération est invariante dans la région $\{3, 4\}$ et pourrait être déplacée dans le segment 2 par exemple. Supposons d'autre part que dans $\{2, 3, 4, 5, 6\}$, A et B dépendent de calculs exécutés dans la région; dès lors, (OP; A; B) ne pourrait plus être déplacée car elle n'est plus invariante dans cette région et nous ne considérons pas le problème du déplacement d'instructions d'un segment d'une région dans un autre segment de la région. Il en résulte que, dans le cas d'une exécution fréquente de $\{3, 4\}$, il aurait été préférable de prendre la liste R_1 comme référenciel pour l'optimisation plutôt que les deux autres listes. En effet, en choisissant R_1, (OP; A; B) aurait pu être déplacée dans une région moins fréquemment exécutée.

2) Nous pouvons transformer le graphe en un graphe équivalent(*) obtenu en dupliquant le segment (3) en un segment (3') relié aux autres segments de la façon suivante :



Remarquons que, pour ce graphe transformé, nous obtenons la liste R suivante:
$\{6\}$, $\{6, 5\}$, $\{4, 3'\}$, $\{2, 3, 5, 6\}$, $\{2, 3, 3', 4, 5, 6\}$.

(*) Nous dirons que deux graphes sont équivalents si les programmes qu'ils représentent sont sémantiquement équivalents. Nous ne préciserons pas plus cette notion d'équivalence sémantique de deux graphes.

Nous présentons ci-dessous un algorithme permettant de réaliser la transformation. Cette transformation de graphe permet, par duplication de segment (partie de code), de déchevaucher des boucles.
Son intérêt peut, a priori, sembler mineur, vu que la majorité des langages de programmation interdisent le chevauchement de boucles. Ceci est vrai pour les boucles traditionnelles (DO, <u>for</u>, ...), il n'est toutefois pas impossible de trouver de tels chevauchements.

Nous sommes pour l'instant dans l'incapacité de fournir une démonstration correcte de l'exactitude de la transformation, pour la simple raison que ce travail est limité dans le temps.

Soit un graphe G. Soient deux régions R_1 et R_2 de G telles que $R_1 \cap R_2 = \{ S_1, \ldots, S_m \}$, $R_1 \cap R_2 \neq R_1$, $R_1 \cap R_2 \neq R_2$.

Ces deux régions sont dites chevauchantes; la transformation a pour but de transformer le graphe G en un graphe dans lequel R_1 et R_2 ne sont plus chevauchantes.

Après transformation du graphe G, une des régions R_1 et R_2 comportera tous les sommets dupliciés, cette région portera le nom de <u>région dupliciée R_d</u>. L'autre région sera appelée la <u>région fixe R_f</u>.

(1) Parmi R_1 et R_2, choisissons la région qui contiendra les sommets dupliciés (exemple $R_1 \rightarrow R_f$ et $R_2 \rightarrow R_d$).

(2) Initialisons R_f et R_d suivant le choix fait en (1) ($R_f = R_1$, $R_d = R_2$); ($R_f \cap R_d = \{ S_1, \ldots, S_m \}$).

(3) $i := 1$.

(4) Choisissons S_i dans $R_f \cap R_d$.

(5) Dupliçons S_i en S'_i.

(6) ($\forall S_k \in R_d$) (<u>si</u> (S_k, S_i) est un arc de G <u>alors</u> créons (S_k, S'_i) et supprimons (S_k, S_i) de G) ($S_k \neq S_i$).

(7) ($\forall S_e$ un sommet de G) (<u>si</u> (S_i, S_e) est un arc de G <u>alors</u> créons (S'_i, S_e)) ($S_e \neq S_i$).

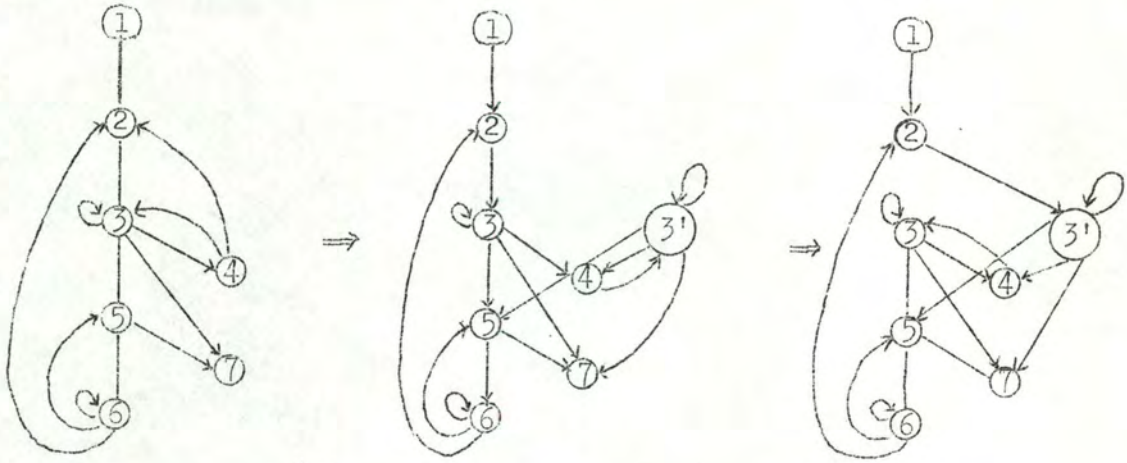
(8) <u>Si</u> (S_i, S_i) est un arc de G <u>alors</u> créons (S'_i, S'_i).

(9) $R_d = (R_d \setminus \{ S_i \}) \cup \{ S'_i \}$.

(10) $i := i+1$; <u>si</u> $i > m$ <u>alors</u> terminons l'algorithme
<u>sinon</u> poursuivons en (4).

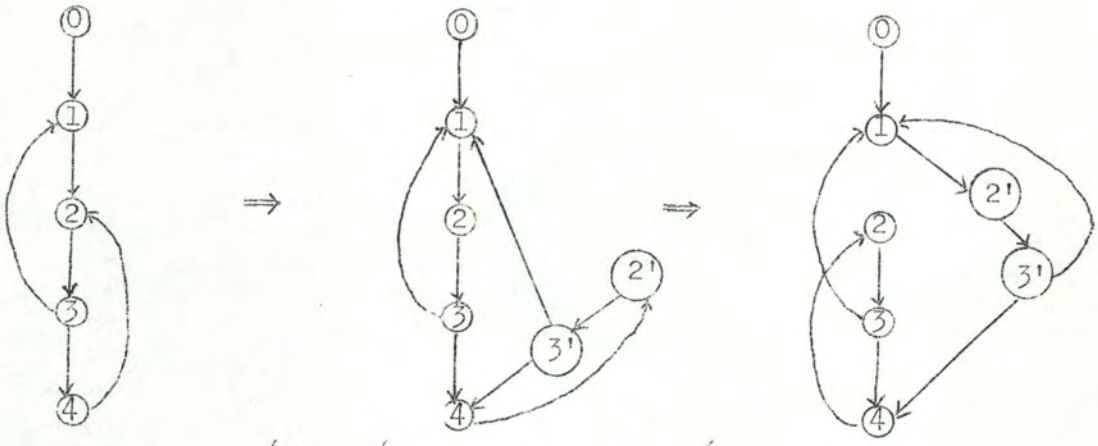
Exemple :

(1)



transformé pour $\begin{cases} R_d = 3 - 4 \\ R_f = 2 - 3 - 5 - 6 \end{cases}$ et pour $\begin{cases} R_f = 3 - 4 \\ R_d = 2 - 3 - 5 - 6 \end{cases}$

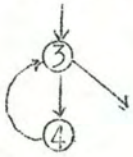
(2)



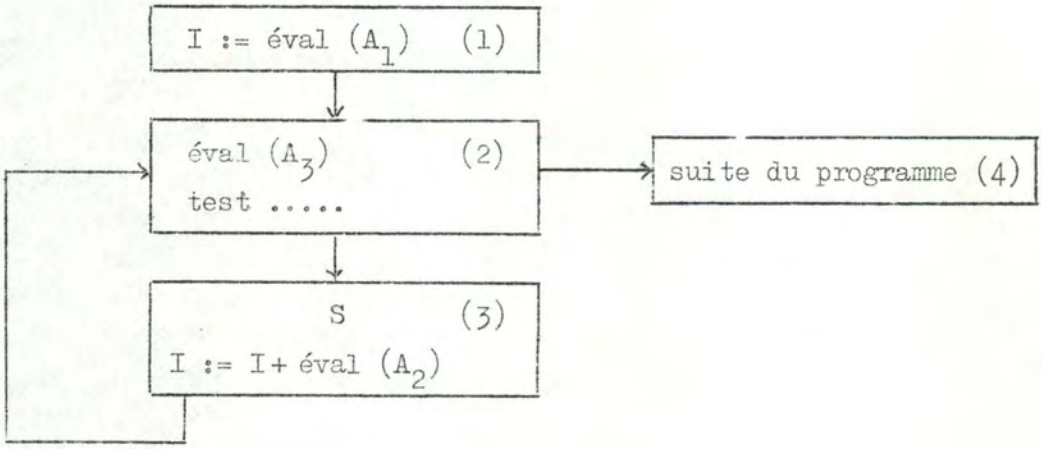
transformé pour $\begin{cases} R_d = \{2, 3, 4\} \\ R_f = \{1, 2, 3\} \end{cases}$ et pour $\begin{cases} R_f = \{2, 3, 4\} \\ R_d = \{1, 2, 3\} \end{cases}$

Remarques

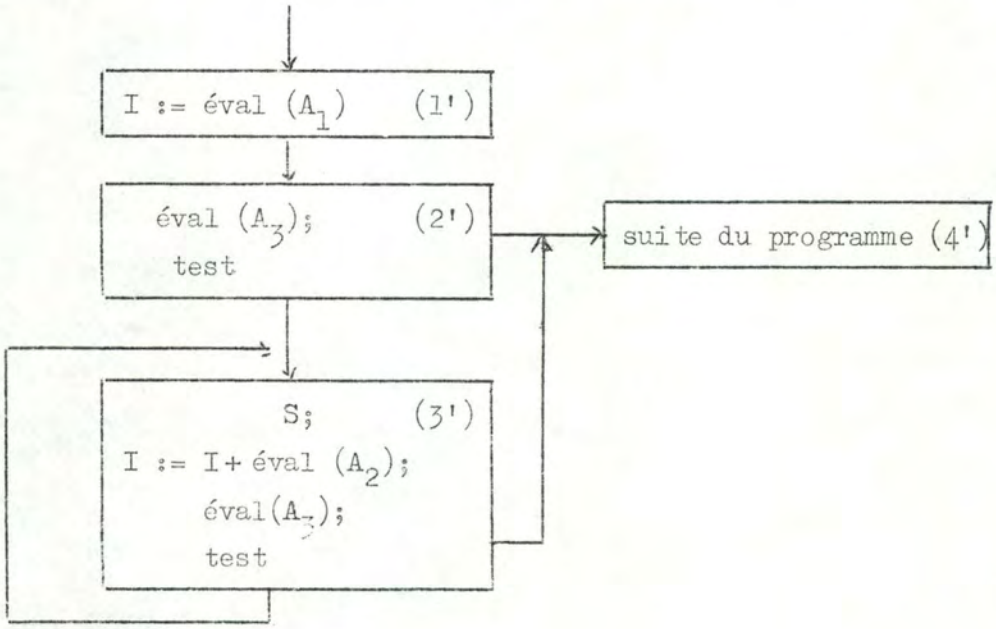
1) Le graphe suivant est très représentatif des boucles <u>for</u> Algol 60.



En effet, soit la boucle <u>for</u> I := A_1 <u>step</u> A_2 <u>until</u> A_3 <u>do</u> S; où A_1, A_2 et A_3 sont des expressions arithmétiques et S une instruction Algol 60; elle peut être représentée par le graphe suivant :

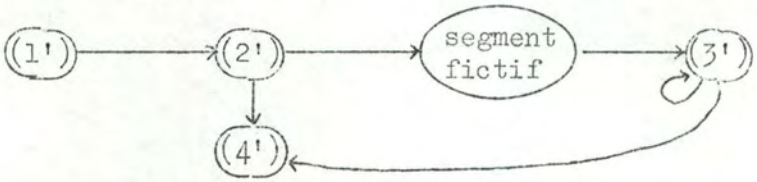


Il est intéressant de transformer ce graphe en son équivalent suivant:



Ceci pour deux raisons essentielles :

1° si nous trouvons, dans S, une assignation du type V := A.E. (Arithmetic Expression) dont les opérandes sont invariants dans la boucle du premier modèle, elle ne peut pas être déplacée dans le segment (1) ou dans le segment (2); tandis que dans le deuxième modèle elle peut être déplacée dans un segment fictif placé entre le segment (2') et (3').



2° En Algol 60, le corps du <u>for</u> (<u>for</u> statement) peut très bien ne jamais être exécuté, il suffit pour cela que l'expression booléenne $(I - A_3) * \text{sign}(A_2) > 0$ soit toujours vrai . Ceci signifie que, si des opérations sont déplacées, dans le modèle 1, vers le segment (1), elles seront exécutées inutilement et provoqueraient des erreurs si elles étaient des assignations ou tout autre type d'opération modifiant la valeur d'une quantité, ce qui n'est pas le cas avec le deuxième modèle.

Remarquons que (1') et (2') pourraient être fusionnés.

Il est souhaitable que les routines sémantiques du compilateur produisent pour un langage tel qu'Algol 60 le deuxième modèle de graphe.

La solution que nous avons présentée ci-dessus n'est certainement pas meilleure que la précédente en ce sens qu'elle augmente le volume du programme. Sans doute permettra-t-elle de diminuer le temps d'exécution du programme car, sur la liste des régions, pourront figurer plus de boucles qu'avec la première solution.

3) Il faudrait s'assurer de ce que les régions les plus visitées du programme figurent dans la liste R. Généralement, les régions les plus visitées sont les boucles internes du programme [KNUTH 71] et celles-ci offrent une structure assez particulière en ce sens qu'elles n'ont jamais qu'une seule entrée et un seul prédécesseur, le segment dans lequel est initialisée la variable de contrôle. Ainsi, si eux régions ont une int rsection non vide et sont telles que l'une ne couvre pas l'autre, nous choisirons celle qui ne possède qu'une seule entrée et un seul prédécesseur.

<u>Exemple</u>

Il y a ici trois régions :

R_1 {1, 2, 3}
R_2 {2, 3, 4}
R_3 {1, 2, 3, 4}

Par notre règle, nous choisirons la région R_2 et nous aurons dès lors la liste R suivante :

R_2, R_3.

Remarquons qu'avec la règle de transformation du graphe, nous pouvions placer sur la liste R les trois régions.

3.1.5. Procédure de décomposition d'un graphe en régions et constitution d'une liste de régions

Pour construire la liste des régions, nous procédons en deux étapes.

En premier lieu, nous construisons une liste P de toutes les régions du programme. Dans une région fortement connexe, tout sommet est son propre successeur et prédécesseur et il existe, dans la région, un circuit passant par ce sommet. Les régions se trouvant sur la liste P sont ordonnées suivant les longueurs des plus longs circuits élémentaires qu'elles contiennent.

En second lieu, à partir de la liste P, nous construisons la liste des régions en utilisant une des règles (1), (2) ou (3) du paragraphe 3.1.4.

3.1.5.1. Rappel de quelques théorèmes

Si C est la matrice booléenne associée au graphe G, la puissance booléenne C^e est la matrice booléenne associée au graphe G_e, lequel possède un arc (i,j) si et seulement s'il existe dans G un chemin de longueur e allant de i à j. G et G_e ont même ensemble de sommets.
Dès lors, si $C^e[i,i] = 1$, il existe, dans G, un circuit de longueur e passant par le sommet i.

La matrice booléenne (*) $B = \sum_{i=1}^{n} C^i$ (somme booléenne des puissances booléennes de C) est telle que $B[i,j] = 1$ si et seulement s'il existe un chemin allant de i à j.

3.1.5.2. Construction de la liste P

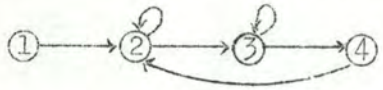
Etant donné un programme divisé en segments d'instructions numérotés 1, 2, ..., n et G son graphe associé, représenté par une matrice booléenne C, la liste P est construite de la façon suivante.

1) La liste P est une liste de vecteurs booléens: $P^i_{[j]} = 1$ si le segment j appartient à la région i, $P^i_{[j]} = 0$ dans le cas contraire.

(*) n est l'ordre du graphe.

2) Si C [i,i] = 1, alors le segment i est à la fois un prédécesseur immédiat et un successeur immédiat de lui-même. Par conséquent, il existe un circuit réduit à une boucle passant par i. Pour ce circuit, nous créons une entrée dans la liste P. ($\forall$ i = 1, ..., n).

Exemple



C	1	2	3	4
1	0	1	0	0
2	0	1	1	0
3	0	0	1	1
4	0	1	0	0

	1	2	3	4
P^1	0	1	0	0
P^2	0	0	1	0

3) $e := 2$.

4) Prenons la puissance booléenne e de C. Si C^e [i,i] = 1, alors il existe un circuit σ de longueur e passant par le segment i. Si le segment j se trouve sur le même circuit que le segment i, alors C^e [j,j] = 1. Inversément, si C^e [i,i] = 1 et Ce [j,j] = 1, on ne peut pas conclure que les segments i et j se trouvent sur un même circuit σ de longueur e. Pour que ceci soit vrai, il faut imposer des conditions supplémentaires, à savoir :

1° il doit exister un chemin de longueur e_1 allant de i à j,

2° il doit exister un chemin de longueur e_2 allant de j à i,

3° $c_1 + e_2$ doit être égal à e.

Si nous acceptions que cette somme soit supérieure à e, nous trouverions un circuit non mentionné par la matrice C^e.

Afin de séparer les différents circuits mentionnés par la matrice C^e, il faut que nous disposions de renseignements supplémentaires.

5) Pour ce faire, nous introduisons une matrice entière D telle que D [i,j] contient la longueur du chemin le plus court allant du segment i au segment j.

La matrice D est mise à jour de la façon suivante :
si C^e [i,j] = 1 et D [i,j] = 0, alors D [i,j] est forcé à e.

6) Lorsque C^e [i,i] devient égal à 1, nous envisageons la possibilité de créer une région comprenant le segment i et tous les segments j tels que

i et j se trouvent sur un même circuit de longueur inférieure ou égale à e.

Cette région est créée dans un vecteur booléen PP; PP [j] = 1 si le segment j appartient à la région en construction, PP [j] = 0 dans le cas contraire (j=1, ..., n).

<u>Si</u> C^e [i,i] = 1, <u>alors</u>

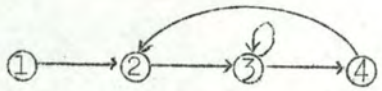
1° posons PP [i] = 1

2° pour tout $j \neq i$ ($1 \leq j \leq n$),

<u>si</u> D [i,j] $\neq$ 0, D [j,i] $\neq$ 0 et D [i,j] + D [j,i] $\leqslant$ e

<u>alors</u> PP [j] = 1;

<u>Exemple</u>



$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \boxed{1} & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad C^2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & \boxed{1} & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

$$C^3 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & \boxed{1} & 1 & 1 \\ 0 & 1 & \boxed{1} & 1 \\ 0 & 0 & 1 & \boxed{1} \end{bmatrix} \quad D = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 3 & 2 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

C^3 [2,2] = 1, d'où PP [2] = 1

j = 1

D [2,1] = 0, D [1,2] = 1 et D [2,1] + D [1,2] = 1 $\leqslant$ 3

d'où PP [1] := 0

j = 3

D [2,3] = 1, D [3,2] = 2 et D [2,3] + D [3,2] = 3 $\leqslant$ 3,

d'où PP [3] := 1

j = 4

D [2,4] = 2, D [4,2] = 1 et D [2,4] + D [4,2] = 3 $\leqslant$ 3,

d'où PP [4] = 1

ainsi PP = $\{2, 3, 4\}$ ou sous forme booléenne

PP = 0111

7) PP est ajouté à la liste P s'il n'y figure pas encore.

Ici nous pourrions faire intervenir la règle suivante :

PP ne sera ajouté à la liste P que s'il est une région fréquemment visitée du programme .

Pour pouvoir appliquer cette règle, il faut bien entendu disposer d'informations relatives à cette fréquence de visite.

8) e := e + 1.

<u>si</u> e > n, <u>alors</u> terminons la procédure

<u>sinon</u> allons en (4).

<u>Remarques</u> : 1. Procédure pour élever une matrice booléenne à une puissance supérieure :

$$C^e * C = C^{e+1}$$

a) Initialisons C^{e+1} à zéro.

b) Pour chaque ligne i de C^e, soit $C^e\,[i,\emptyset]$, et pour chaque élément $C^e\,[i,j]$ différent de zéro dans cette ligne, on fait $C^{e+1}\,[i,\emptyset] := C^{e+1}\,[i,\emptyset] \vee C^e\,[j,\emptyset]$.

Cette procédure est très rapide si l'opérateur d'"oring" est très rapide.

Cet algorithme découle de l'idée suivante :

1) C^{e+1} donne tous les chemins de longueur e+1 allant d'un segment i vers un segment j.

2) Un chemin de longueur e+1 est formé d'un sous-chemin de longueur e et d'un arc.

3) Ainsi si $C^e\,[i,j] = 1$, il existe un chemin de longueur e allant de i à j et il suffit de regarder quels sont les arcs qui quittent ce sommet j pour trouver les chemins de longueur e+1.

<u>Exemple</u>

$$\Longrightarrow C^e\,[i,j] = 1 \Longrightarrow \begin{cases} C^{e+1}\,[i,k] = 1 \\ C^{e+1}\,[i,m] = 1 \\ C^{e+1}\,[j,n] = 1 \end{cases}$$

3.1.5.3. Construction de la liste R de régions

La procédure 3.1.5.2 nous délivrait une liste $P \equiv P_1, P_2, \ldots, P_n$ de circuits élémentaires possibles ans le graphe G et tell. que

$\forall\, i,j \quad \{1, \ldots, n\} \; i < j \quad P_i \neq P_j$ et longueur$(P_i) \leq$ longueur (P_j).

A partir de la liste P, nous construisons la liste R de la façon suivante:

1) Plaçons P_1 sur la liste R.

2) Si pour P_i (i=2, ..., n), les conditions 1 et 2 de la définition de la liste R sont respectées, alors plaçons P_i sur la liste R.

S'il existe P_j tel que $j < i$ et $P_j \quad P_i \neq \emptyset$ et $P_j \cap P_i \neq P_j$, alors appliquons une des règles (1), (2) ou (3) de 3.1.4 et plaçons sur la liste R la ou les régions choisies.

Exemple

Pour l'exemple du paragraphe 3.1.1, la liste P est la suivante :

	1	2	3	4	5	6	7	8	9	10
P^1	0	0	0	0	0	0	0	1	0	0
P^2	0	0	0	1	1	0	0	0	0	0
P^3	0	1	1	0	0	1	1	0	0	0
P^4	0	1	1	1	1	0	1	0	0	0
P^5	0	1	1	1	1	1	1	0	0	0

1) Plaçons P^1 sur la liste de régions.

2) $P^1 \cap P^2 = \emptyset$ et $P^1 \neq P^2$ d'où P^2 peut être placé sur la liste de régions.

3) $P^1 \cap P^3 = \emptyset$ et $P^1 \neq P^3$
$P^2 \cap P^3 = \emptyset$ et $P^2 \neq P^3$
d'où P^3 peut être placé sur la liste de régions.

4) $P^1 \cap P^4 = \emptyset$ et $P^1 \neq P^4$
$P^2 \cap P^4 = P^2$ et $P^2 \neq P^4$
mais $P^3 \cap P^4 \neq \emptyset$ et $\neq P^3$; dès lors, P^4 ne peut pas être placé sur la liste de régions.
Nous appliquons une des règles (1), (2) ou (3) du paragraphe 3.1.4, par exemple la règle (1): nous enlevons donc P^3 de la liste de régions.

5) $P^5 \cap P^1 = \emptyset$ et $P^1 \neq P^5$
$P^5 \cap P^2 = P^2$ et $P^2 \neq P^5$
d'où P^5 peut être placé sur la liste de régions.

6) Finalement, la liste de régions est la suivante :

	1	2	3	4	5	6	7	8	9	10
R_1	0	0	0	0	0	0	0	1	0	0
R_2	0	0	0	1	1	0	0	0	0	0
R_3	0	1	1	1	1	1	1	0	0	0

3.2. Quelques concepts supplémentaires

3.2.1. Notion de prédominance et son utilisation

Soit un graphe G ayant un seul sommet d'entrée (si ce n'est pas le cas, nous construirions un sommet fictif d'entrée).

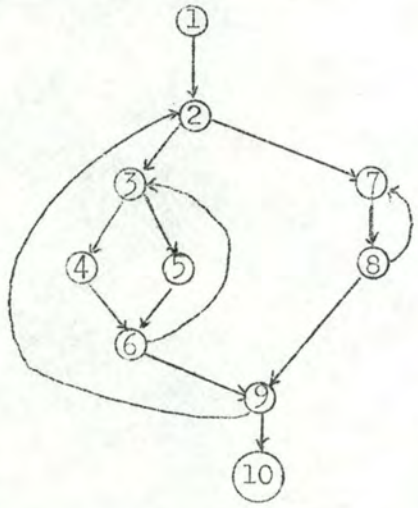
Prédominateur d'un sommet

Un sommet x_i prédomine un sommet x_j dans un graphe G si et seulement si x_i appartiebt à tous les chemins allant de l'entrée du graphe vers x_j. On dit que x_i est un prédominateur de x_j.

Prédominateur immédiat d'un sommet

Un sommet x_i est un prédominateur immédiat d'un sommet x_j dans un graphe G si et seulement s'il est un prédominateur de x_j et s'il n'existe pas d'autre prédominateur de x_j sur tout chemin allant de x_i à x_j.

Exemple



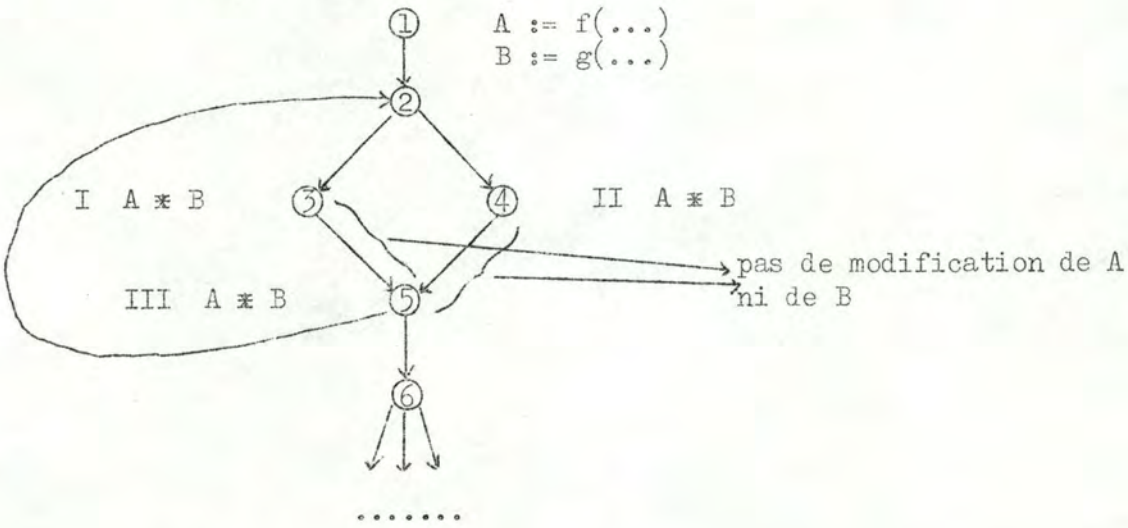
- 1 est un prédominateur immédiat de 2.
- 3 est un prédominateur immédiat de 4, 5 et 6.
- 2 et 1 prédominent 3 et 9.
- 7 est un prédominateur immédiat de 8.
- 2 est un prédominateur immédiat de 9.

Cette notion de prédominateur a été développée par Medlock et Lowry pour le compilateur optimiseur IBM 360/OS FORTRAN-H.

Cette relation de prédominance est transitive et nous pouvons affirmer que si i et j prédominent un même segment k, alors soit i prédomine j, soit j prédomine i. Plus généralement, si un segment k est prédominé par plusieurs segments $i_1, i_2, \ldots, i_n$, il existe parmi ces segments un segment i_j ($i_j \in \{1, \ldots, n\}$) qui est prédominé par tous les autres. i_j est le prédominateur immédiat de k.

Cette relation de prédominance permet de généraliser la technique de la suppression des instructions redondantes. En effet, si le segment i prédomine le segment j, nous pouvons affirmer que , quel que soit le chemin suivi, lors de l'exploitation, le segment i sera toujours exécuté avant le segment j. Dès lors, si l'opération A * B se trouve à la fois dans les segments i et j et si, sur les chemins qui vont de i à j, ni A ni B ne voient leur valeur modifiée, nous pouvons conclure que l'opération A * B figurant dans le segment j est redondante.

Lowry et Medlock ont développé cette technique dans le compilateur 360/OS FORTRAN-H. Elle peut sembler plus performante que notre technique de suppression d'instructions redondantes, mais elle n'est toutefois pas parfaite; en effet, dans l'exemple suivant, l'instruction III du segment 5 ne sera pas reconnue comme étant redondante.



Pour amélicrer cette technique, il faudrait lui adjoindre un système permettant de déplacer, à l'intérieur d'une boucle, des instructions d'un segment vers un segment qui le prédomine.

Exemple - Ce système déplacerait les instructions I et II dans le segment 2.

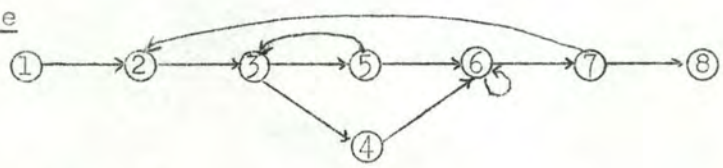
Pour plus d'information au sujet de cette technique dont l'idée fondamentale a été décrite ci-dessus, nous donnons maintenant la référence. "Object Code Optimization", Edward S. LOWRY and C.W. MEDLOCK (IBM Corporation, New York), CACM Volume 12, Number 1, january 1969, pages 13, 22.

3.2.2. Notion d'"articulation" dans une région

Le concept de point d'articulation est intéressant lorsque des instructions invariantes sont déplacées d'une région.

On dit qu'un segment i est "un point d'articulation" d'une région R s'il se trouve sur tous les chemins allant d'une entrée de la région R vers une sortie de R.

Exemple



- dans la région $\{6\}$, 6 est un point d'articulation;
- dans la région $\{3,5\}$, 5 est un point d'articulation;
- dans la région $\{2, 3, 5, 6, 4, 7\}$, 2, 3, 6, 7 sont des points d'articulation.

La notion de point d'articulation est intéressante. En effet, i étant un point d'articulation d'une région R, on peut affirmer que si la région R est visitée lors de l'exploitation du programme, i sera certainement exécuté.

3.2.3. Procédure pour trouver les points d'entrée et de sortie d'une région R

3.2.3.1. Les points d'entrée

Les points d'entrée d'une région R sont les successeurs immédiats des prédécesseurs immédiats de la région R. Ce principe est à la base de la procédure qui nous permet de trouver les points d'entrée de R.

Les prédécesseurs immédiats de la région R sont les prédécesseurs immédiats des sommets de R et se trouvant à l'extérieur de R. Ce principe est à la base de la procédure qui nous permet de trouver les prédécesseurs immédiats de R.

(1) Soient le graphe G à n sommets numérotés de 1 à n et la région R représentée sous la forme d'un vecteur booléen. Soit C la matrice booléenne associée à G.

(2) Soit PR le vecteur booléen des prédécesseurs immédiats des sommets appartenant à R.

$$PR = \bigvee_i \left\{ C\,[\emptyset, i] \;\middle|\; R(i) = 1 \right\}$$

PR est l'union des colonnes de C correspondant à des sommets appartenant à R.

(3) Soit PRI le vecteur booléen des prédécesseurs immédiats de la région R. On a :

$$PRI = PR \wedge \overline{R}$$

(4) Soit PE le vecteur booléen des points d'entrée de la région R. On a :

$$PE = \left[\vee \left\{ C\,[i, \emptyset] \;\middle|\; i \text{ tel que } PRI\,[i] = 1 \right\} \right] \wedge R$$

3.2.3.2. <u>Les points de sortie</u>

Les points de sortie d'une région R sont les prédécesseurs immédiats des successeurs immédiats de la région R.

Les successeurs immédiats de R sont les successeurs immédiats des sommets de R et se trouvent à l'extérieur de R.

Ces deux principes nous permettent de trouver les points de sortie et les successeurs immédiats d'une région R.

(1) Soient le graphe G à n sommets numérotés de 1 à n, la région R, représentée sous la forme d'un vecteur booléen et la matrice booléenne C associée à G.

(2) Soit SR le vecteur booléen des successeurs immédiats des sommets appartenant à R. On a :

$$SR = \bigvee_i \left\{ C\,[i, \emptyset] \;\middle|\; R(i) = 1 \right\}$$

SR est l'union des lignes de C correspondant à des sommets appartenant à R.

(3) Soit SRI le vecteur booléen des successeurs immédiats de la région R.

$$SRI = SR \wedge \overline{R}$$

(4) PS le vecteur booléen des points de sortie de R.

$$PS = [\vee \{ C[\emptyset, i] \mid SRI[i] = 1 \}] \wedge R$$

3.2.3.3. Exemples

Prenons la région 2 - 3 - 4 - 5 - 6 - 7 de l'exemple du paragraphe 3.1.1 :

	1	2	3	4	5	6	7	8	9	10
R =	0	1	1	1	1	1	1	0	0	0

C	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	1	0	0	0
6	0	0	0	0	0	0	1	1	0	0
7	0	1	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

$[\vee \{ C[\emptyset\ i] \mid SRI(i) = 1 \}]$

	1	2	3	4	5	6	7	8	9	10
PR =	1	1	1	1	1	1	1	1	0	0
$\overline{R}$ =	1	0	0	0	0	0	0	1	1	1
PRI =	1	0	0	0	0	0	0	0	0	0
$[\vee \{ C[i,o] \mid PRI(i) = 1 \}]$ =	0	1	0	0	0	0	0	0	0	0
PE =	0	1	0	0	0	0	0	0	0	0
SR =	0	1	1	1	1	1	1	1	1	0
$\overline{R}$ =	1	0	0	0	0	0	0	1	1	1
SRI =	0	0	0	0	0	0	0	1	1	0
$[\vee \{ C[\emptyset\ i] \mid SRI(i) = 1 \}]$ =	0	0	0	0	0	1	1	1	0	0
PS =	0	0	0	0	0	1	1	0	0	0

3.2.4. Constitution de la liste des points d'articulation

S'il n'y a qu'un seul point d'entrée dans une région R, alors il doit être un point d'articulation.

S'il n'y a qu'un seul point de sortie dans une région R, alors il doit être un point d'articulation.

Si, dans la région R, le point de sortie est unique et confondu avec l'unique point d'entrée, alors ce segment est l'unique point d'articulation.

Dans le cas général, lorsqu'il y a plusieurs points d'entrée et (ou) plusieurs points de sortie, ou lorsque le point d'entrée unique n'est pas

confondu avec l'unique point de sortie, la procédure à suivre est la suivante.

Soient le graphe G constitué des segments $S_1, \ldots, S_n$ et la matrice booléenne C associée à G.

Soit R la région constituée des sommets $S_1, \ldots, S_m$ $(m \leq n)$.

Soient PS la liste booléenne des points de sortie de R et PE la liste booléenne des points d'entrées de R.

(o) $i := 1$

(1) Nous construisons la matrice associée au sous-graphe correspondant à la région R et dans laquelle nous supprimons le sommet S_i :

- les colonnes et les lignes de C correspondant à des sommets de G ne faisant pas partie de R sont annulées;
- la colonne et la ligne correspondant à S_i sont annulées.

Soit C^* cette nouvelle matrice booléenne.

(2) Nous construisons ensuite la matrice booléenne B de la façon suivante :

$$B := \sum_{i=1}^{n} C^{*i}$$

Il s'agit d'une somme booléenne de puissances booléennes de C^*.

Cette matrice B est telle que $B[S_i, S_j] = 1$ s'il existe un chemin allant de S_i à S_j dans le sous-graphe associé à C^*.

(3) <u>Si</u> pour tout S_j $(j=1,\ldots,m)$ tel que $PE[S_j] = 1$, $B[S_j, \emptyset] \wedge PS = 0$, <u>alors</u> S_i doit être un point d'articulation puisqu'il n'existe pas de chemin allant d'une entrée vers une sortie qui ne passe pas par le sommet S_i. Listons S_i dans la liste LPA des points d'articulation de R.

(4) $i := i + 1$.

<u>Si</u> $i > m$, <u>alors</u> terminons l'algorithme,

<u>sinon</u> allons en (1).

<u>Remarque</u> : La matrice booléenne construite à l'étape (1) a la forme

$$C^* = \left(\begin{array}{c|c|c} 0 & 0 & 0 \\ \hline 0 & U & 0 \\ \hline 0 & 0 & 0 \end{array}\right) \quad \text{d'où} \quad B = \left(\begin{array}{c|c|c} 0 & 0 & 0 \\ \hline 0 & V & 0 \\ \hline 0 & 0 & 0 \end{array}\right)$$

avec $$V = \sum_{i=1}^{n} U^i.$$

Il suffira donc de calculer V pour connaître B.

3.3. Dépendance des items de données dans l'ensemble d'un programme *

A cause de l'existence de boucles dans un programme, la dépendance des items de données ne peut pas être représentée, comme nous l'avions fait pour un segment d'instructions, par des arbres de dépendance. Il est pourtant important de connaître les liens existant entre une assignation d'une valeur à une quantité et une utilisation de cette variable dans un programme.

Nous appellerons Lien Assignation-Référence un chemin du programme allant d'une instruction modifiant la valeur d'une quantité vers une utilisation de cette valeur assignée à la quantité.

Un tel lien peut être représenté par un ensemble ordonné de segments $LAR_v = (S_A, S_1, \ldots, S_n, S_R)$ où S_A est le segment contenant l'instruction modifiant la valeur de la quantité v, S_R le segment contenant l'utilisation de la quantité v, S_1 un successeur immédiat de S_A, S_i un prédécesseur immédiat de S_{i+1}, S_n un prédécesseur immédiat de S_R et tel que, dans aucun des segments S_i, il n'y ait d'instruction modifiant la valeur de v (bien qu'ils puissent contenir d'autres utilisations de v).

Ces liens sont importan car toute coupure de ces liens par l'optimisation produira, dans la majorité des cas, un programme sémantiquement non équivalent au programme non optimisé.

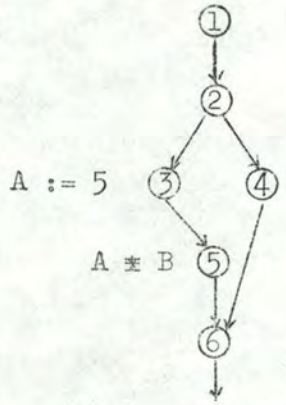
Il y a plusieurs façons de couper un lien LAR.

1° Un lien LAR peut être risé lorsque nous déplaçons soit l'instruction modifiant la valeur de la quantité, soit l'utilisation en dehors de la portée du lien. Ceci aura pour conséquence qu'il pourra exister dans le programme un chemin allant de l'entrée vers l'utilisation sans nécessairement passer par l'instruction modifiant la quantité.

---

* Dans ce paragraphe, assignation sera parfois utilisé pour les mots "instructions modifiant la valeur d'une quantité".

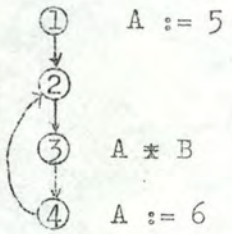
Exemple



Le déplacement de A * B en 6 brise le lien LAR_A = (3 , 5).
Effectivement, si A * B est déplacé en 6 il existe un chemin 1 - 2 - 4 - 6 ne passant pas nécessairement par A := 5.

2° En interférant avec le lien LAR, par exemple en introduisant une nouvelle instruction modifiant la valeur de la quantité dans le chemin allant de l'assignation vers l'utilisation.

Exemple



Le déplacement A := 6 dans 1 interfère avec le lien LAR_A = (1 , 2 , 3)

Les techniques développées tâcheront de ne pas couper les liens L.A.R., même si ceux-ci ne sont pas explicitement mémorisés. En effet, une telle tâche serait fastidieuse et demanderait une analyse fort étendue dans le temps. La seule information globale sur les assignations (instructions modifiant la valeur d'une quantité) et sur les utilisations sera donnée sous forme de tables de vecteurs booléens.
Nous aurons ainsi trois tables:ASS pour les assignations, REF pour les utilisations et RAA pour les utilisations précédant des assignations dans un segment d'instructions.

a) La table ASS

Il y a une entrée dans cette table pour chaque quantité du programme. Soit V une variable ou un identificateur de tableau (nom de tableau):
ASS [V,i] = 1 si la variable V est modifiée par une instruction dans le segment i, ASS [V,i] égale zéro dans le cas contraire.

b) La table REF

De même que pour ASS, il y a dans cette table une entrée pour chaque variable du programme et chaque tableau. Soit V une variable ou un identificateur de tableau (nom de tableau); REF [V,i] = 1 si la quantité V est utilisée dans le segment i, REF [V,i] = 0 dans le cas contraire.

c) La table RAA

De même que pour les deux précédentes, il y a dans cette table autant d'entrées que le programme comporte de variables et de tableaux.

RAA [V,i] = 1 si la quantité V est utilisée dans le segment i et que cette utilisation figure dans le segment avant une instruction I modifiant la valeur de cette quantité, RAA [V,i] vaut zéro dans le cas contraire. L'instruction V peut ne pas exister dans le segment.

Nous ne discuterons pas ici du problème de l'organisation de ces tables. Il nous semble en tout cas qu'une organisation dans laquelle l'accès se fait au moyen d'une clé (ici l'identificateur de variable ou de tableau) est préférable. Pour un programme volumineux, ces tables occupent une place non négligeable. Cela explique que l'analyse du flot des données dans un programme est abordé différemment dans les recherches actuelles. On trouvera dans ce travail une étude théorique de l'analyse du flot des données dans un programme.

Les trois tables peuvent être construites par les routines sémantiques du compilateur ou, si l'on désire garder une indépendance totale vis-à-vis de la phase d'optimisation, par les procédures de FOLDING ou de "Suppression des instructions redondantes".

Nous sommes peut-être un peu draconien pour les tableaux, plus partivulièrement pour les variables indicées. En effet, I := 5; A [I] := 20; ne modifie que la cinquième composante du vecteur A; or nous disons que, par la constitution des tables, A [I] := 20 modifie tout le tableau. Nous pourrions éviter cet inconvénient mais le coût d'une amélioration serait très élevé, car au lieu de considérer globalement un tableau comme une entité modifiable par toute instruction modifiant un élément, nous serions obligé d'individualiser chaque élément du tableau. Ainsi les tables ASS, RAA, REF seraient énormes pour un programme traitant des tableaux de taille moyenne.

Chapitre IV

DEPLACEMENT D'INSTRUCTIONS INVARIANTES

Le déplacement d'instructions invariantes est la troisième technique d'optimisation de programme que nous envisageons.

Dans ce chapitre, nous développerons une définition constructive d'instructions invariantes et de constantes de région. Nous montrerons comment, par un choix adéquat de l'organisation de l'optimiseur, on peut réduire cette technique globale à une technique qui n'abuse pas trop de l'information globale.

L'ensemble de ce chapitre est fortement inspiré de l'article d'Allen [ALLEN 69]. Nous avons apporté cependant quelques développements, notamment une règle de choix basée sur la notion de niveau étendu dans un graphe, dont on trouvera une définition dans les Annexes de ce travail.

— § —

4.1. Définition d'instructions invariantes

4.1.1. Définition d'une constante de région

Une quantité (variable simple, variable indicée, tableau) est dite constante dans une région R et est appelée constante de région

si et seulement si

il n'existe, dans la région R, aucune instruction modifiant la valeur de la quantité.

Cette définition générale doit être particularisée pour les différents types d'opérandes que nous pouvions rencontrer dans la forme intermédiaire. Nous allons donner ici ces définitions particulières de façon constructive, c'est-à-dire en utilisant les tables dont nous disposons ASS, REF et RAA.

Soit $R = S_1, S_2, \ldots, S_n$ la Région.

4.1.1.1. Une constante est par définition même une constante de région. Nous supposons, bien sûr, qu'une constante ne peut jamais être modifiée par une instruction. Si c'était le cas, il faudrait traiter les constantes comme l'on traite des variables simples.

4.1.1.2. Une variable simple V sera une constante de région si et seulement si

$$\prod_{i=1}^{n} \text{ASS}\left[V, S_i\right] = 0 \quad (\prod = \text{produit booléen})$$

4.1.1.3. Un tableau T sera une constante de région si et seulement si

$$\prod_{i=1}^{n} \text{ASS}\left[T, S_i\right] = 0$$

4.1.1.4. Une variable indicée V [SUBS] sera une constante de région si et seulement si

1° le tableau V est une constante de Région,

2° si SUBS est quantité constante dans la région R.

Nous n'envisageons pas dans cette définition le cas où SUBS est une référence à une instruction intermédiaire car nous n'avons pas jusqu'ici défini ce qu'était une instruction invariante dans une région R.

4.1.2. Instructions invariantes dans une région R

D'une façon générale, on dit qu'une instruction intermédiaire I située dans une région R est invariante dans cette région si et seulement si ses opérandes ne sont modifiés par aucune instruction de la région.
Pour être quelque peu plus précis et pour montrer l'utilité de nos tables, nous allons séparer la définition en deux définitions relatives respectivement au cas des instructions intermédiaires ayant des opérandes qui ne sont que des quantités (variables simples, variables indicées, tableaux) et au cas des instructions intermédiaires ayant parmi leurs opérandes des instructions intermédiaires.

4.1.2.1. Instructions invariantes ayant pour opérandes des quantités uniquement

Nous allons ici envisager les différents types d'instructions intermédiaires pouvant avoir des opérandes qui sont des quantités.

4.1.2.1.1. Les opérations intermédiaires du type "Computation"

Soit R ≡ S_1, ..., S_n la région.
Soit I ≡ (OP; ARG 1; ARG 2) l'opération intermédiaire.
I est invariante dans R

si et seulement si

1° ARG 1 est constante de région
2° ARG 2 est constante de région.

4.1.2.1.2. Les instructions du type "Assignation"

Soit I ≡ (:=; ARG 1; ARG 2) l'assignation intermédiaire.
I est invariante dans R

si et seulement si

ARG 2 est une constante de région.

4.1.2.1.3. Les instructions du type "Indexed Assignation"

Soit I ≡ (:=; T_1 [SUBS 1] ; ARG 2) l'assignation indexée.
I est invariante dans R

si et seulement si

1° ARG 2 est une constante de région
2° SUBS 1 est une constante de région.

4.1.2.1.4. Instruction du type APS

Soit I ≡ (APS; PS; ARG 2) l'instruction du type APS.

I est invariante dans R

si et seulement si

ARG 2 est une constante de région.

4.1.2.1.5. Instruction du type AP

Comme nous l'avons déjà souligné précédemment, un appel de procédure peut être considéré comme étant à la fois une utilisation de résultats antérieurs et une modification de certaines quantités.
Les quantités modifiables sont les paramètres actuels et les opérandes globaux. Pour être plus précis, il faudrait pouvoir déterminer quels paramètres actuels et quels opérandes globaux sont modifiés par la procédure. Des paramètres appelés par valeur ou par argument postiche (Dummy Argument) ne seront certainement pas modifiés par la procédure. Quant aux autres types d'appels, les paramètres actuels réellement modifiés sont plus difficiles à déterminer. Nous supposerons que les opérandes globaux sont toujours modifiés par la procédure. Il est très difficile, dans les langages du type Algol 60, de déterminer exactement la liste des opérandes globaux d'une procédure quelque peu complexe. Ces considérations, qui n'ont peut-être pas de rapport étroit avec notre propos, montrent comment il faut agir sur les tables ASS, REF et RAA.

Soit I ≡ $\left[AP; \left\{ \begin{matrix} P \\ ARG\ 1 \end{matrix} \right.; (P_1, P_2, \ldots, P_n)\right]$ l'instruction de type AP avec $PG_1, \ldots, PG_m$ comme paramètres globaux.

I est dite invariante

si et seulement si

1° $P_1, P_2, \ldots, P_n$ sont des constantes de région

2° $PG_1, \ldots, PG_m$ sont des constantes de région.

4.1.2.2. Instructions invariantes ayant pour opérandes des instructions intermédiaires

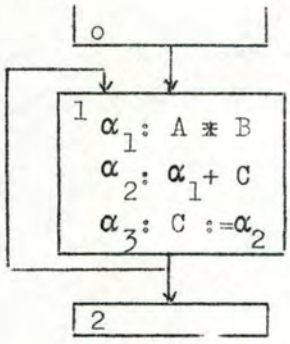
Si l'opérande d'une instruction intermédiaire est une instruction intermédiaire I_o, il faut vérifier si I_o est une instruction invariante dans la région R pour pouvoir conclure à l'invariance de I dans R.

Cette analyse nous obligera peut-être à remonter très loin dans un segment pour décider de l'invariance d'une instruction. C'est pour cette raison que des méthodes tenant compte des particularités de notre texte intermédiaire seront développées dans la suite.

Nous ne referons pas ici l'analyse des différents types d'instructions. Cette analyse est dénuée d'intérêt, vu qu'elle est identique à celle du paragraphe 4.1.2.1.
Nous rappellerons que cette analyse ne doit pas traiter les instructions du type AP.

4.2. Critères généraux de déplacement

Une instruction I invariante dans une région R n'est pas toujours déplaçable, c'est-à-dire telle que l'on puisse la déplacer vers des régions moins visitées du programme. Ainsi dans l'exemple suivant :



il est impossible de déplacer α_1 : A * B dans le segment 2, même si A et B sont des constantes de région.

Nous décrirons dans ce paragraphe les conditions générales - c'est-à-dire celles qui ne sont pas particulières à notre forme intermédiaire - pour qu'une instruction invariante soit déplaçable.

4.2.1. Le déplacement d'une instruction ne peut pas avoir pour effet de déplacer son résultat au-dessus d'une utilisation de ce résultat

Dans le cas de l'exemple précédent, l'instruction α_1: A * B est déplaçable dans le segment o mais pas dans le segment 2.

4.2.2. Le déplacement d'une instruction ne peut pas causer d'interférences avec un lien LAR existant.

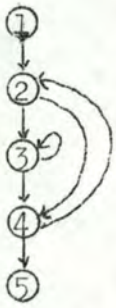
Ceci peut arriver lorsque l'instruction que l'on déplace est une instruction modifiant la valeur d'une quantité et qu'elle est déposée dans un segment faisant partie d'un lien LAR relatif à cette quantité.

Exemple

A := 3

A := 4

X := A



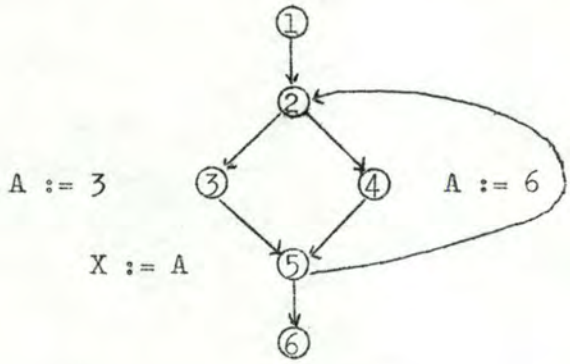
L'instruction A := 4 ne peut être déplacée nulle part.

En effet :

- en déplaçant A := 4 dans le segment 4, on interfère avec le lien $LAR_A = (2,4)$,
- en déplaçant A := 4 dans le segment 2, on interfère avec le lien $LAR_A = (2,4)$.

4.2.3. Le déplacement d'une instruction ne peut pas briser un lien LAR existant.

Plus généralement, on peut dire que le déplacement d'une instruction ne peut pas briser un lien entre cette instruction et l'utilisation de son résultat.



Aucune des instructions A := 3 et A := 6 ne peut être déplacée de la région $\{2, 3, 4, 5\}$; en effet:

- si l'on déplace A := 3 dans le segment 1, X := A sera mal défini sur le chemin 1-2-4-5-2-3-5 car X aura la valeur 6 alors que normalement X devrait contenir la valeur 3.

- si l'on déplace A := 6 dans le segment 1, un raisonnement analogue peut être tenu pour le chemin 1-2-3-5-2-4-5.

- La règle 4.2.1 nous empêche de déplacer A := 3 ou A := 6 vers le segment 6 .

4.2.4. Avec la définition d'instruction invariante que nous avons prise, nous posons implicitement une quatrième condition. En effet, dire que les opérandes d'une instruction ne peuvent être modifiés par aucune instruction de la région revient à exiger que l'on ne puisse pas déplacer une instruction au-dessus d'une instruction modifiant la valeur d'un de ses opérandes.

4.3. Définitions de concepts supplémentaires

4.3.1. Définition et utilisation d'une instruction intermédiaire

Une définition d'une instruction intermédiaire est l'occurrence de cette instruction intermédiaire dans un segment.

Exemple TI		TSQ
α_1(*; A; B)	a)	α_1
α_2(:=; C; α_1)	b)	α_2
α_3(+; A; B)	c)	α_3
α_4(:=; D; α_3)	d)	α_4

a est la définition de α_1, b est la définition de α_2, etc...

Une utilisation d'une instruction intermédiaire est l'occurrence de cette instruction intermédiaire comme opérande d'une autre instruction intermédiaire.

Exemple

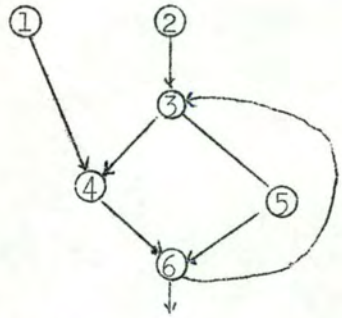
Dans l'exemple précédent, α_2 est une utilisation de α_1, α_4 est une utilisation de α_3.

4.3.2. Déplacement en arrière et en avant

Nous appelons déplacement en arrière d'une instruction I le déplacement de I vers tous les segments prédécesseurs immédiats d'une région R.

Exemple

I = (OP; A1; A2)

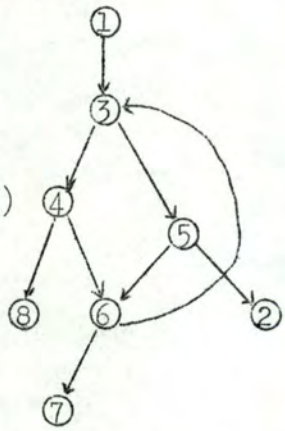


Supposons que I soit déplaçable. Le déplacement de I dans 2 et 1 est un déplacement en arrière.

Nous appelons déplacement en avant d'une instruction I le déplacement de I vers tous les segments successeurs immédiats d'une région R.

Exemple

I = (OP; A1; A2)



Supposons que I soit déplaçable. Le déplacement de I dans 2, 7 et 8 est un déplacement en avant.

4.4. Particularités dues à la forme intermédiaire choisie et à l'organisation de l'optimiseur

Par une organisation adéquate de l'optimiseur et par un choix adéquat de la forme intermédiaire utilisée, le déplacement d'instructions qui requiert une information globale sur le programme peut être réalisé sur base d'informations locales tout en respectant les conditions globales citées dans le paragraphe 4.2.

C'est ce que nous montrerons dans ce paragraphe.

4.4.1. Remarques préliminaires

Il est important que l'élimination d'instructions redondantes se réalise avant le déplacement d'instructions invariantes.

Ceci est dû à plusieurs raisons. L'une, essentielle, résulte des définitions d'instruction redondante et d'instruction invariante. Les autres sont particulières à la structure des procédures réalisant les différentes techniques d'optimisation. Elles ne pourront être mises en évidence que lorsque nous aurons envisagé l'ensemble de la technique de déplacement des instructions.

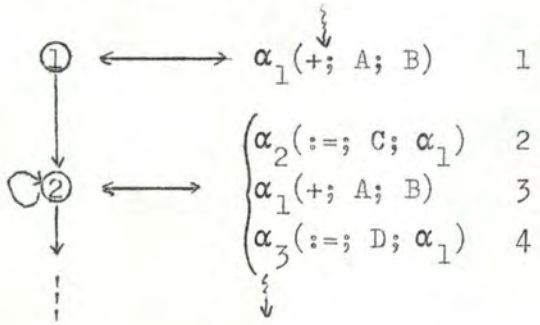
Pour mettre en évidence la première raison, nous nous servirons d'un exemple. Considérons le segment suivant formant une région R dans le graphe G.

1) $\alpha_1(+;\ A;\ B)$
2) $\alpha_2(:=;\ C;\ \alpha_1)$
3) $\alpha_1(+;\ A;\ B)$
4) $\alpha_3(:=;\ D;\ \alpha_1)$

segment ②

Si A et B sont des constantes de région, les instructions 1 et 3 sont invariantes dans la région 2 .

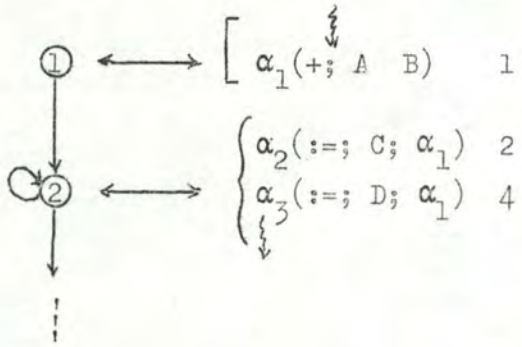
Si l'élimination des instructions redondantes n'était réalisée qu'après le déplacement d'instructions, nous aurions le résultat suivant.



En effet, quoiqu'invariante dans la région, l'instruction 3 n'est pas déplaçable.

Par contre, si l'élimination des instructions redondantes est accomplie

avant le déplacement d'instructions, nous avons le résultat suivant.



En effet, l'instruction 3 a d'abord été éliminée du programme puisqu'elle est redondante; ensuite, l'instruction 1 a été déplacée.

4.4.2. Caractéristiques du texte intermédiaire après la procédure d'élimination des instructions redondantes

Afin d'obtenir les caractéristiques que nous énoncerons ci-dessous, il est important que l'élimination des instructions redondantes soit terminée avant que l'on passe au déplacement d'instructions.

4.4.2.1. Les instructions d'un segment d'instructions sont ordonnées par numéro de niveau croissant.

4.4.2.2. Une instruction intermédiaire du type COMPUTATION, APS ou AP (c'est-à-dire représentant un résultat intermédiaire) est toujours utilisée par au moins une instruction intermédiaire. Chacune de ces utilisations possède un numéro de niveau au moins supérieur d'une unité au numéro de niveau de la définition.

4.4.2.3. Toute instruction intermédiaire de type COMPUTATION, APS ou AP, utilisée dans un segment, est également définie dans ce segment. Le numéro de niveau de la définition est inférieur strictement au numéro de niveau de l'utilisation.

C'est cette troisième caractéristique qui nous permet d'affirmer que le déplacement d'instructions peut être réalisé sur la base d'informations locales (orientée segment).

4.4.2.4. S'il existe plusieurs définitions d'une même instruction intermédiaire dans un segment, celles-ci possèdent des numéros de niveau

différents. En effet, s'il n'en était pas ainsi, elles auraient été déclarées redondantes.

4.4.3. Conséquences des caractéristiques énoncées en 4.4.2.

De ces caractéristiques, nous pouvons déduire les observations suivantes relativement au déplacement en arrière d'instructions. Pour déterminer les instructions déplaçables en arrière dans un segment, nous examinons le segment depuis l'entrée jusqu'à la sortie.

4.4.3.1. Si n est le numéro de niveau maximum des numéros de niveau de toutes les instructions d'un même segment qui ont été déplacées en arrière jusqu'à présent, alors seules les instructions ayant un numéro de niveau inférieur ou égal à n+1 sont candidates à un déplacement en arrière

En effet, soit I une instruction de numéro de niveau égal à n+1, I est une utilisation d'instruction de niveau inférieur ou égal à n. Or, n est le numéro de niveau le plus élevé des instructions actuellement déplacées; par conséquent, parmi les instructions qui sont déplacées, certaines peuvent être des opérandes de I, d'où I est candidate à un déplacement en arrière. Le raisonnement ci-dessus est certainement valable pour les instructions I de niveau inférieur à n+1.

Cette conclusion est très importante car elle nous permettra de gagner en efficacité lors de la recherche d'instructions candidates au déplacement en arrière.

Initialement, seules les instructions de niveau 1 seront candidates à un déplacement en arrière. Si aucune d'entre elles n'est déplaçable, alors aucune instruction du segment ne pourra être déplacée. De plus, si toutes les instructions de niveau n ne sont pas déplaçables, a fortiori toutes celles de niveau supérieur ne le seront pas.

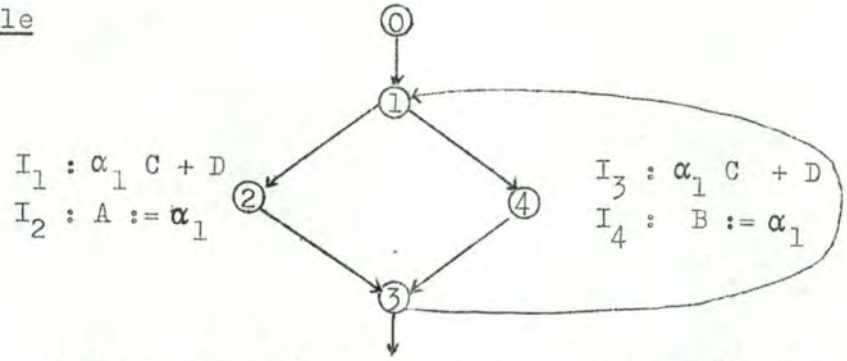
4.4.3.2. Lorsque l'on examine une instruction intermédiaire I afin de déterminer si elle peut être déplacée en arrière, il ne faut considérer que les opérandes de cette instruction et non pas ses utilisations

En effet, en raison de 4.4.2.1, les utilisations de I se trouvent après I dans le segment, dès lors la condition 4.2.1 ne peut jamais être violée. Mais il pourrait se trouver des utilisations de I dans d'autres

segments, c'est ces dernières qu'il ne faut pas considérer.

Ceci permet de déplacer des instructions qui pourraient apparaître comme non déplaçables.

Exemple



Si C et D sont des constantes de région ($\{1,2,3,4\}$), nous pouvons déplacer en arrière l'instruction I_1 dans le segment 0 sans qu'il soit nécessaire de reconnaître l'utilisation I_2 de I_1. Si ensuite nous désirons déplacer en arrière I_3, nous devrions tenir compte du problème du déplacement d'une instruction au-dessus d'une utilisation dans I_2, de son résultat (chemin (4, 3, 1, 2)).

Il est important de remarquer que ceci réduit l'environnement de recherches pour déterminer si une instruction est déplaçable ou non.

Il faut remarquer également qu'il ne faut pas considérer les définitions précédentes de l'instruction I; en effet, celles-ci doivent avoir un numéro de niveau inférieur et si actuellement, elles n'ont pas pu être déplacées vers l'arrière, c'est que leurs opérandes soit ne sont pas des constantes de région, soit ne sont pas déplaçables. Par conséquent, en examinant les opérandes de I, on doit arriver aux mêmes conclusions.

4.4.3.3. Pour déterminer si une définition d'une instruction intermédiaire I opérande d'une autre instruction intermédiaire J figure ou ne figure pas dans une région, il suffit de chercher cette définition dans les instructions précédant l'instruction J dans le segment qui la contient.

En effet, nous savons par les caractéristiques 4.4.2.2 et 4.4.2.3 que seules les définitions figurant dans le segment peuvent affecter l'utilisation.

Si, dans les instructions qui précèdent l'utilisation, nous ne trouvons pas de définition de l'instruction utilisée, cela implique que cette définition était déplaçable et a été déplacée en arrière. Par conséquent, l'utilisation présente est candidate au déplacement en arrière.

De ces trois observations 4.4.3. (1-2-3), nous pouvons conclure que certaines décisions de déplacement pourront être prises avec une information locale. Toutefois, pour les quantités du programme et les quantités générées par le compilateur, il faut une information plus globale. Celle-ci est obtenue au moyen des tables ASS, REF, RAA.

Certains résultats restent valables pour les déplacements en avant, notamment le suivant (on commence par examiner les dernières instructions du segment pour respecter le principe 4.2.1), sans que l'on change quoi que ce soit à la forme intermédiaire ou aux procédures adoptées.

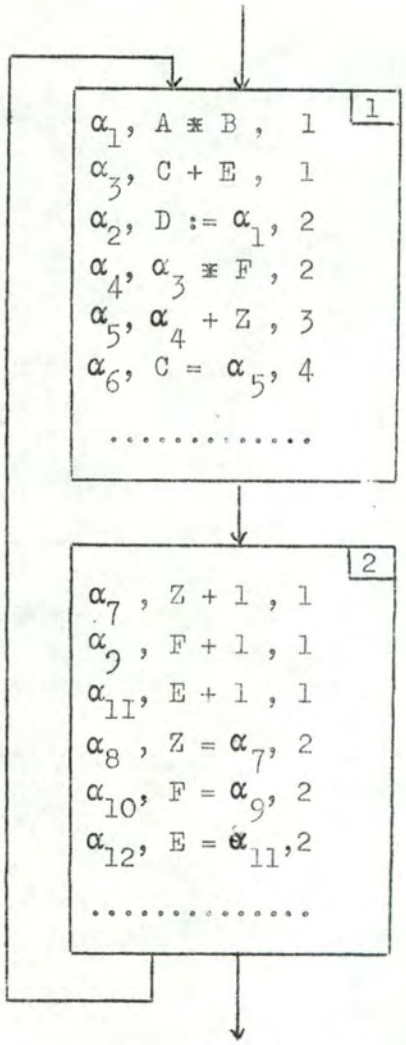
<u>Pour déterminer si une utilisation d'une instruction intermédiaire I figure, ou ne figure pas, dans une région, il suffit de chercher cette utilisation dans les instructions qui suivent l'instruction définissant I.</u>

Si l'on ne trouve pas d'utilisation de I, cela signifie que ces utilisations ont été déplacées vers l'avant et que la définition présente est, elle aussi, candidate au déplacement vers l'avant.

Ce résultat est le 4.4.3.3 précédent, adapté au déplacement en avant. Les autres résultats ne sont plus valables tels quels.

On pourrait penser à l'énoncé suivant de 4.4.3.1 pour les déplacements en avant: si n est le numéro de niveau minimum parmi les numéros de niveau de toutes les instructions déplacées en avant jusqu'à présent, alors seules les instructions ayant un numéro de niveau supérieur ou égal à n-1 sont candidates à un déplacement en avant. Ce résultat n'est plus vrai car certaines instructions de niveau $n-m > 0$ peuvent être candidates au déplacement en avant alors qu'aucune instruction de niveau n ou de niveau n-1 n'est déplaçable en avant.

Exemple



Examinons le segment 1. Ni α_5 ni α_6 ne sont déplaçables en avant. Or, α_1 et α_2 le sont, si l'on suppose que

1° D n'est pas utilisé dans la région,

2° D n'est pas redéfini dans la région,
(remodifié)

3° α_1 n'est redéfini ni dans le segment 1 ni dans le segment 2,

4° A et B ne sont pas redéfinis sur tout chemin allant vers la sortie de la région.

On est donc obligé de "scanner" tout un segment et d'examiner toutes les instructions de ce segment, même si aucune d'entre elles n'est déplaçable vers l'avant. L'analyse requise sera donc assez longue.

En essayent d'adapter la deuxième conclusion relative aux déplacements en arrière, pour les déplacements en avant, on pourrait penser à

l'énoncé suivant :

> lorsque l'on examine une instruction intermédiaire I afin de déterminer si elle peut être déplacée en avant, il ne faut considérer que les opérandes de cette instruction et non pas ses définitions.

Ceci n'est plus entièrement valable. Il faut exiger en plus que les utilisations de l'instruction intermédiaire soient prises en considération.

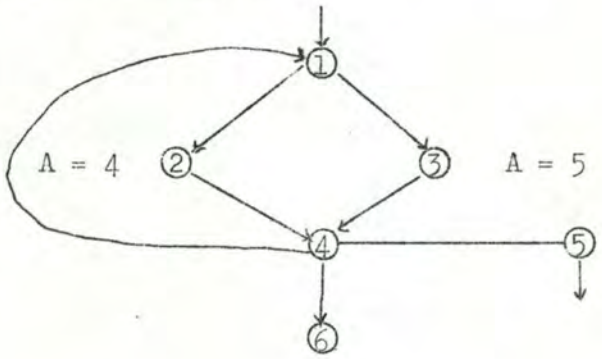
Nous examinons maintenant l'influence de la notion de région fortement connexe sur la relation entre des utilisations de quantités et des modifications de quantités (instructions modifiant la valeur des quantités) :

1) si une quantité est utilisée dans une région sans qu'elle y soit modifiée, alors nous supposons que la quantité possède une valeur déterminée lorsque nous entrons dans la région;

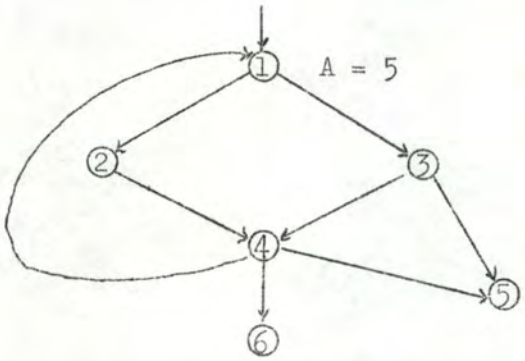
2) si une quantité est à la fois modifiée et utilisée dans une région, alors toute modification peut être reliée à n'importe quelle utilisation, à cause de la propriété de connexité forte;

3) si une quantité est modifiée dans une région, on ne peut généralement pas supposer que la valeur attribuée à la quantité par la modification, est la valeur de la quantité lorsque nous entrons dans un successeur immédiat quelconque de la région.

<u>Exemple</u>



Lorsque nous entrons dans ⑥ , A est soit égal à 4, soit égal à 5. C'est également vrai lorsque nous entrons dans ⑤ .
Par contre, dans l'exemple suivant, la valeur de A peut être déterminée de façon exacte si l'on suppose que A n'est modifié ni dans ②, ni dans ③, ni dans ④.



Le deuxième exemple montre qu'il y a des exceptions. Ainsi, des quantités modifiées dans des points d'articulation sont généralement des exceptions.

4.5. Critères de déplacement particularisés

Pour décrire ces critères, nous distinguerons deux types d'instructions :

1° Les instructions de calcul, c'est-à-dire celles du type "COMPUTATION", "APS" et "AP" (appel de fonction).

2° Les instructions définissant des quantités, c'est-à-dire modifiant la valeur de quantités.

Nous ne nous attarderons pas aux instructions du type "AP". Comme nous l'avons déjà dit précédemment, elles méritent toute une étude à elles seules.

4.5.1. Critères pour déplacement en arrière

4.5.1.1. Les instructions de calcul (COMPUTATION ou APS)

Pour qu'une telle instruction soit déplaçable, il suffit (la partie SUBS d'une variable indicée étant considérée comme un opérande) que

1° ses opérandes qui sont des instructions intermédiaires ne soient pas définis précédemment dans le segment;

2° ses opérandes qui sont des quantités soient des constantes de région.

4.5.1.2. Instruction modifiant des quantités (définition d'une quantité). (ASSIGNATIONS et INDEXED ASSIGNATIONS)

Pour qu'une telle instruction soit déplaçable, il suffit (la partie SUBS d'une variable indicée est considérée comme un opérande) que :

1° ses opérandes qui sont des instructions intermédiaires ne soient pas définies précédemment dans le segment;

2° ses opérandes qui sont des quantités soient des constantes de région;

3° le segment dans lequel se trouve l'instruction soit un point d'articulation.

Nous devons nous assurer de ce que la quantité possède, dans toute la région, la valeur que lui attribue l'instruction, puisque nous déplaçons l'instruction dans tous les segments prédécesseurs immédiats de la région.

4° De plus, il faut qu'il n'existe pas d'autres définitions ou d'utilisations de la quantité sur tout chemin allant d'une entrée de la région vers l'instruction.

En effet; s'il existait une utilisation de la quantité entre une entrée dans la région et l'instruction par le déplacement de l'instruction modifiant la quantité, nous interférerions avec un lien LAR existant entre l'utilisation et une définition se trouvant en dehors de la région. Pour l'existence d'une autre définition, un raisonnement semblable mènerait au critère 4°; il s'agit ici d'une coupure de lien LAR.

4.5.2. Critère pour déplacement en avant

L'idée de base dans le déplacement en avant est d'éviter qu'une instruction dont les opérandes ne sont pas nécessairement des constantes de région, mais qui n'est pas utilisée dans la région, soit exécutée plusieurs fois inutilement. Pour cela, on la déplace en avant en dehors de la région, de telle façon que les opérandes de l'instruction soient modifiés par les itérations successives dans la région mais que la valeur de l'instruction ne soit calculée qu'une seule fois au moment où l'on sort de la région.

4.5.2.1. Les instructions de calcul

Pour qu'une telle instruction soit déplaçable vers l'avant, il suffit que

1° ses opérandes, qui sont des instructions intermédiaires ne soient pas redéfinis sur tout chemin allant de l'instruction vers une sortie de la région;

2° ses opérandes, qui sont des quantités, ne soient pas modifiés sur tout chemin allant de l'instruction vers une sortie de la région.

Ces deux conditions assurent que l'on n'interfère pas avec un lien LAR existant.

3° L'instruction candidate au déplacement ne soit pas utilisée entre l'instruction et la sortie du segment dans laquelle elle se trouve.

4.5.2.2. Instruction modifiant des quantités

Pour qu'une telle instruction soit déplaçable vers l'avant, il suffit que

1° ses opérandes qui sont des instructions intermédiaires ne soient pas redéfinis sur tout chemin allant de l'instruction vers une sortie de la région;

2° ses opérandes qui sont des quantités ne soient pas modifiés sur tout chemin allant de l'instruction vers une sortie de la région.

Cette condition nous assure que la modification de valeur de la quantité est toujours exécutée.

4° Il n'existe pas d'autres instructions modifiant la valeur de la quantité sur tout chemin allant de l'instruction vers une sortie de la région.

5° La quantité modifiée par l'instruction ne soit pas utilisée dans la région.

Ceci pour ne pas briser un lien LAR.

Il faut remarquer que tous ces critères sont des particularisations des critères généraux 4.2.1, 4.2.2, 4.2.3 et 4.2.4. Pour ne pas allonger exagérément la partie écrite de ce travail, nous n'avons pas repris toutes les justifications.

4.6. Procédure pour le déplacement en arrière

Nous ne donnerons ici que les principes d'une procédure due à Allen et aussi une amélioration, que nous pensons être sensible et qui s'appuie sur la décomposition d'un graphe en niveaux étendus (Annexe 1).

Supposons que les régions R_1, R_2, ..., R_{e-1} aient été traitées par l'optimiseur (FOLDING, Suppression d'instructions redondantes, déplacement d'instructions, etc...).

1) R_e est sélectionnée dans la liste R et l'ensemble $R'_e = R_e \cap (\bigcap_{i=1}^{e-1} \bar{R}_i)$ des segments de R_e qui n'ont pas encore été examinés, est calculé.

Seuls les segments figurant dans R'_e peuvent contenir des instructions déplaçables. En effet, si une instruction I figurant dans un segment $S \in R_e \setminus R'_e$ était déclaré déplaçable lorsque l'on examine R_e, elle le serait a fortiori lorsque l'on examinait les régions R_1, R_2, ..., R_{e-1}.

2) Les tables ASS, REF, RAA tiennent compte de l'ensemble des régions appartenant à la région R_e.

Elles sont générées lorsque les segments $S \in (\bigcup_{i=1}^{e} R_i)$ sont examinés pour l'élimination d'instructions redondantes.

De plus, après que chaque segment de R'_e ait été traité par la procédure de déplacement d'instructions, elles doivent être mises à jour.

Exemple

Si I utilise les quantités v_1 et v_2 dans le segment S, si I est déplacée et si I est la seule instruction utilisant v_1 et v_2 dans S, alors on aura les actions suivantes sur ces tables: REF $[v_1,S]$:= REF $[v_2,S]$:=RAA $[v_1, S]$:= RAA $[v_2,S]$:= 0.

3) Afin de pouvoir utiliser le résultat 4.4.3.1, une table maxlevel comportant une entrée par segment figurant dans R'_e est initialisée à zéro. maxlevel $[S_j]$ contient le numéro de niveau maximum parmi les numéros de niveau de toutes les instructions déplacées de S_j vers l'extérieur de la région.

4) Les points d'articulation, les entrées, les sorties, les prédécesseurs de R_e et ses successeurs sont recherchés.

5) Un segment fictif est initialisé. Il contiendra toutes les instructions qui ont été déplacées de la région R_e. Il sera inséré dans le programme suivant certains critères que nous verrons plus loin, et aussi après certains traitements (Folding, suppression d'instructions redondantes).

6) Dans chaque segment S appartenant à R'_e, on examine chaque instruction ayant un numéro de niveau inférieur ou égal à 1 + maxlevel [S]. Si une telle instruction satisfait aux critères de déplacement en arrière, on la déplace dans le segment fictif et on la supprime du segment S; de plus, si son numéro de niveau est égal à maxlevel [S] + 1, on met maxlevel [S] à jour.

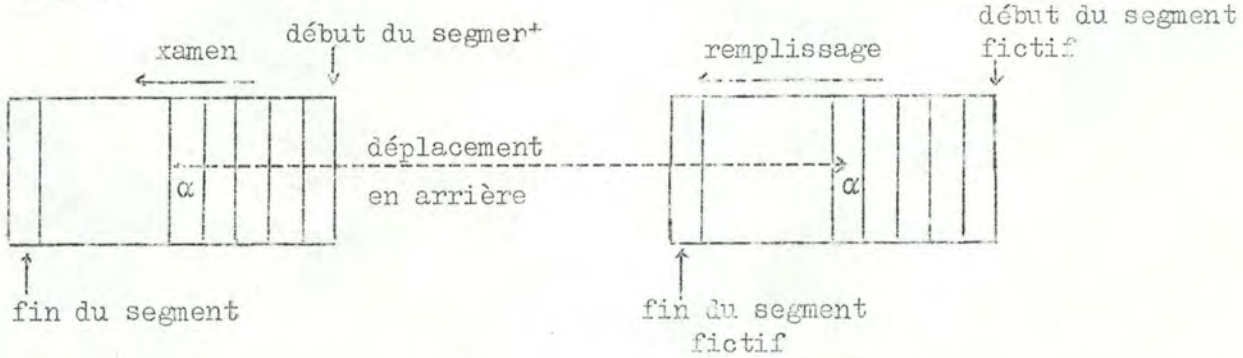
<u>Exemple</u> pour une instruction du type "COMPUTATION" se trouvant en $j^{ème}$ position dans un segment figurant dans TSQ.

a) Pour chacun de ses opérandes qui est une instruction intermédiaire, on cherche, dans les positions 1 jusqu'à j-1 du segment dans TSQ, s'il n'existe pas une définition de ces opérandes.

b) Pour chaque opérande qui est une quantité, on applique à la quantité la condition 4.1.1.1, 4.1.1.2, 4.1.1.3 ou 4.1.1.4 suivant le cas.

Si les deux conditions sont satisfaites, alors l'instruction est déplaçable et on effectue le déplacement.

7) Le déplacement d'instructions peut être visualisé par le schéma suivant.



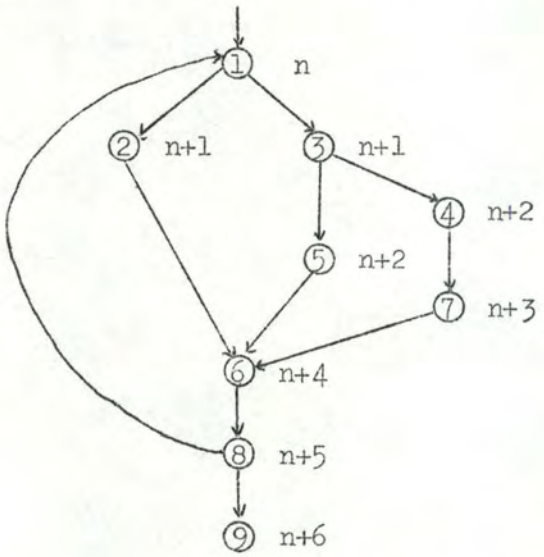
Il faut remarquer qu'il est essentiel que cela se passe ainsi car il faut respecter la notion de séquence.

8) Lorsque tous les segments de R'_e ont été examinés et que des instructions modifiant des quantités ont été déplacées, des instructions de R'_e utilisant ces quantités deviennent candidates au déplacement et doivent

être examinées. C'est ici (pour traiter R'_e de façon complète) que nous pouvons introduire une amélioration pour diminuer le nombre d'appels récursifs de la procédure. En effet, si le graphe du programme a été décomposé en niveaux étendus, nous pouvons énoncer la règle suivante :

> Après avoir traité un segment S appartenant à R'_e, les segments candidats au traitement sont les segments ayant un numéro de niveau étendu égal au numéro de niveau étendu du segment S ou égal au numéro de niveau étendu du segment S, augmenté d'une unité.

<u>Exemple</u>



Les segments candidats après le traitement du segment 3, sont les segments 5, 4, 2.
Si 2 n'a pas été traité, celui qui sera choisi sera le segment 2.

Il faut préciser la règle en introduisant la remarque suivante :

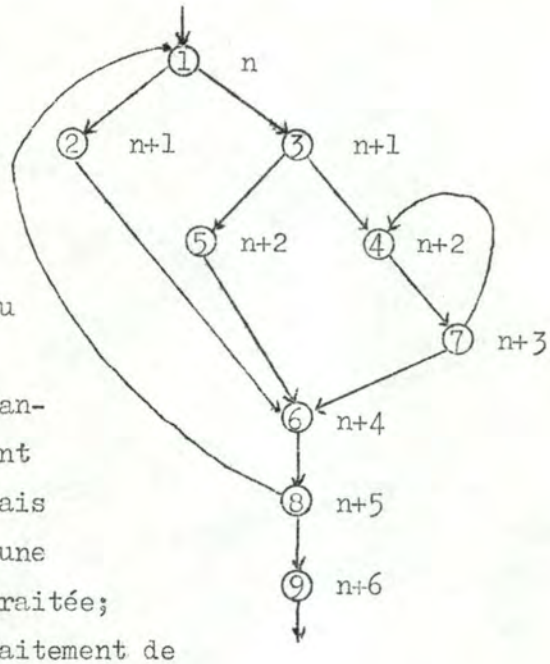
Les segments K ayant un numéro de niveau étendu supérieur d'une unité au numéro de niveau étendu du segment S peuvent faire partie de sous-région de R_e ; dès lors, on choisira le segment qui aura un numéro de niveau le plus proche de celui de S.

x

x x

x

Exemple



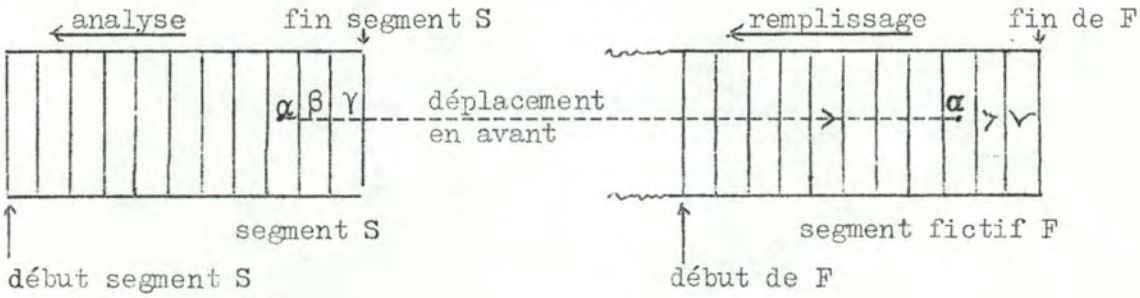
Après traitement du segment ② et du segment ③, les candidats au traitement sont ⑤ et ④, mais ④ fait partie d'une sous-région déjà traitée; dès lors, après traitement de ⑤, on choisira ⑥.

On commence toujours par les segments ayant un numéro de niveau étendu minimal, c'est-à-dire le plus petit de ceux des segments figurant dans R'_e.

9) A la fin de la procédure, le segment fictif contient toutes les instructions qui ont été déplacées vers l'arrière depuis la région R_e. Avant que ce segment ne soit inséré dans le programme, d'autres procédures vont venir insérer des instructions dans ce segment.

4.7. Quelques remarques sur le déplacement en avant

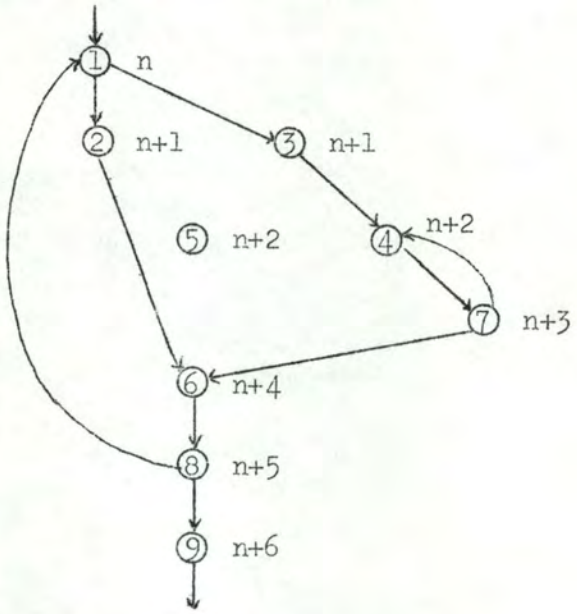
La procédure réalisant le déplacement en avant devra fournir une analyse plus longue avant de décider qu'aucune instruction n'est déplaçable vers l'avant dans un segment. Mais, en gros, cette procédure est analogue à la précédente. Nous pouvons la visualiser par le schéma suivant :



Pour obtenir la règle de choix des segments à traiter, il suffit d'inverser la règle 8 de 3.5.

Exemple

Après traitement de ⑥ pour le déplacement en avant, c'est le segment ⑤ qui est choisi. La procédure de déplacement en avant aura commencé par le segment ⑧.



4.8. Remarque sur la structure de bloc Algol 60

Il existe, en Algol 60, des instructions intermédiaires qu'il faut rendre non déplaçables, par exemple les instructions intermédiaires d'ouverture de blocs (INBLOCK) et de terminaison de blocs (OUTBLOCK).

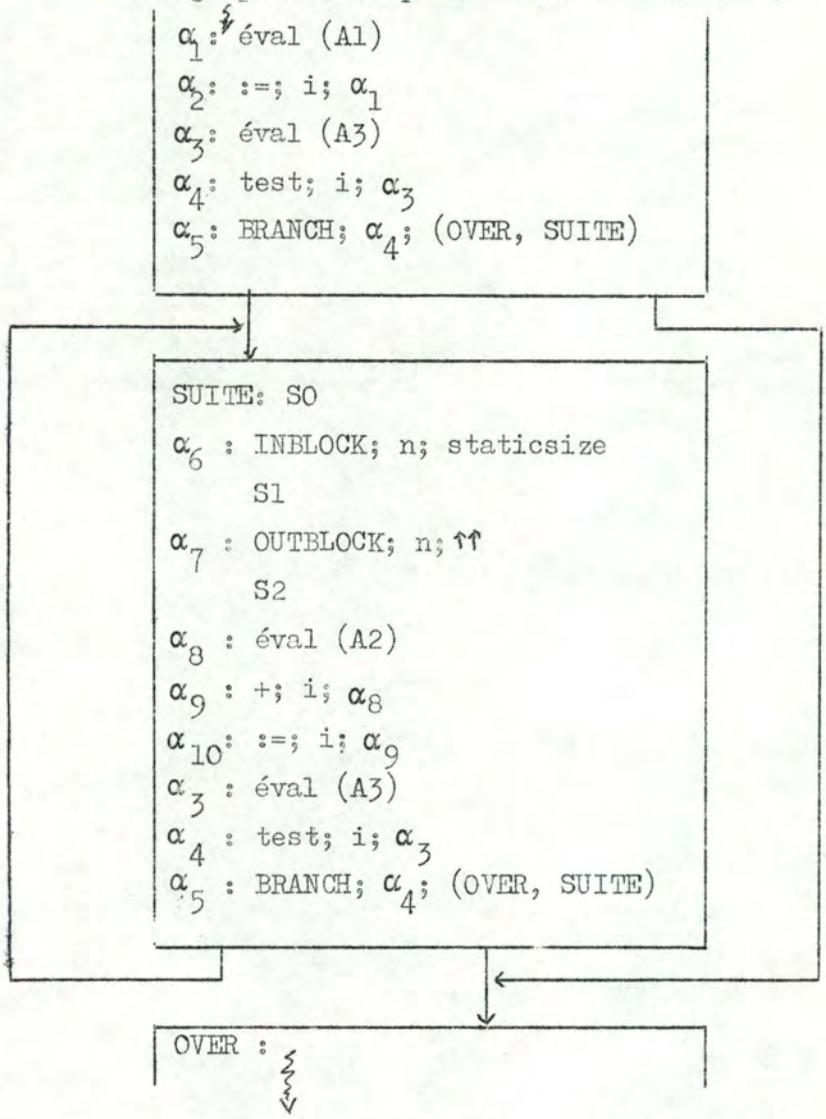
En Algol 60, les quantités sont caractérisées par un "scope" (durée de vie). La durée de vie d'une quantité Algol 60 est égale à la durée de vie du bloc dans lequel elle a été déclarée. Dès lors, il faut veiller, lors du déplacement de code, à ne pas déplacer en dehors de la portée du bloc des instructions utilisant des quantités déclarées dans ce bloc.

Nous illustrons ce dernier point par un exemple :

```
for i := A1 step A2 until A3 do
        begin S0;
              begin integer x;
                    S1;
              end;
              S2;
        end
```

où A1, A2 et A3 sont des expressions arithmétiques et S0, S1 et S2 sont des instructions Algol 60.

Le graphe correspondant est le suivant :



Nous remarquons que s'il existe des instructions déplaçables dans le code de S_1 (par exemple l'instruction I utilisant la quantité x), nous devons tenir compte de la structure de bloc (I ne peut pas être déplacée).

- INBLOCK; n; staticsize; est l'instruction intermédiaire d'entrée dans un bloc Algol 60. n: est le numéro d'imbrication du bloc. Staticsize_n: est la taille de la "Data Area" associée au bloc n.
- OUTBLOCK; n; ↑↑ ; est l'instruction intermédiaire de sortie d'un bloc ALGOL 60 - n: idem que pour INBLOCK.

Chapitre V

REDUCTION D'OPERATEURS A DES OPERATEURS PLUS EFFICIENTS

Le déplacement d'instructions invariantes ne déplace que les instructions invariantes déplaçables. Souvent on a intérêt à déplacer des instructions utilisant des opérateurs lents en dehors de la région. Ceci n'est pas toujours possible, comme il a été exposé dans le chapitre précédent.

La réduction d'opérateurs à des opérateurs plus efficients est une technique qui permettra de déplacer en dehors d'une région certaines opérations utilisant des opérateurs peu rapides et de remplacer ces opérations par d'autres opérations équivalentes mais plus rapides.

Le gain en efficience obtenu est la différence entre le temps d'exécution répétitive de l'instruction déplacée et le temps d'exécution répétitive du groupe d'instructions qui la remplace.

Cette technique n'optimise pas le volume du programme (diminution de la place mémoire occupée) mais son temps d'exécution.

5.1. Quantité récursivement assignée.

5.2. Restriction de la définition 4.1 et les raisons.

5.3. Opérateurs réductibles et non réductibles.

5.4. Instructions réductibles et séquences d'instructions réductibles.

5.5. Utilisation de la notion de séquence d'instructions réductibles.

5.6. Avantage de la notion de séquence d'instructions réductibles et de la réduction d'opérateurs à des opérateurs plus efficients.

5.7. Procédure de réduction des opérateurs.

5.8. Discussion sur la réduction d'opérateurs à des opérateurs plus efficients.

5.9. Optimisation particulière pour les boucles du type DO (FORTRAN IV) ou <u>for</u> (ALGOL 60) transformée.

5.10. Remarques.

La plupart des idées de ce chapitre sont extraites de F.E. Allen [ALLEN 69] et dans une moindre mesure de Davie Gries [GRIES 70].

5.1. Quantité récursivement assignée

Une quantité V est dite récursivement assignée si et seulement si la partie droite de l'assignation est une fonction dépendant de V, c'est-à-dire possède la forme f(V).

Remarquons que si V est une variable indicée, l'expression en partie droite ne peut pas modifier la valeur des indices.

Exemple - A [I,J] := A [I,J] * (B [I] + C [I])
- A = SIN (A)
- A = A + B * C + R

En général, un appel de procédure peut être considéré, pour certaines quantités, comme étant une assignation récursive.

Exemple procedure p(x); integer x; x := x+1;

L'appel p(x) est une assignation récursive.

La définition ci-dessus étant trop générale pour nos besoins, nous en adopterons une autre, plus restrictive, dans le paragraphe suivant .

L'instruction V = f(V) sera ci-après appelée assignation récursive. (Certains auteurs l'appellent définition récursive.)

5.2. Restrictions de la définition 5.1 et les raisons de ces restrictions

5.2.1. Restrictions

Les restrictions que nous imposerons sur des quantités récursivement assignées nous sont dictées par les traitements que nous désirons réaliser sur ces quantités.

Les quantités qui nous intéresseront dans la suite de ce chapitre sont celles qui satisfont aux restrictions suivantes :

1° La quantité est une variable non indicée de type entier;

2° V = f(V) appartient à une région;

3° V = f(V) a la forme V = V + DELTA, où DELTA est soit une constante de région, soit une instruction constante* dans la région. DELTA peut être signé;

* voir page suivante.

4° Toutes les instructions modifiant la valeur de V dans la région doivent être des assignations récursives satisfaisant à la condition 3°.

Exemples I := I + (-5); I := I + 5; I := I + 3 * K + J;
I := J + I

Dorénavant, lorsque nous parlerons de variable récursivement assignée, il faudra comprendre une variable vérifiant les quatre conditions précédentes.

Remarques Les seules formules possibles pour DELTA sont les suivantes :

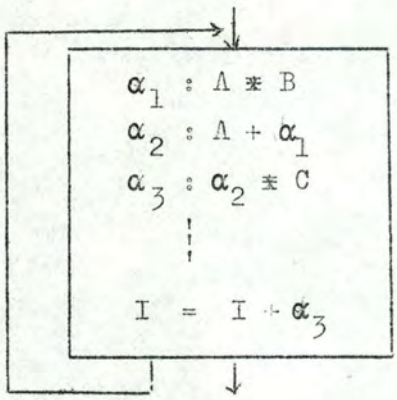
(1°) une constante;

(2°) une variable simple qui est une constante de région;

(3°) une variable indicée dont les indices sont des constantes de région et telle que le tableau dont elle est un élément soit également une constante de région;

(4°) une instruction dont les opérandes sont soit des constantes de région, soit des instructions intermédiaires constantes dans la région.

Exemple



C, A et B étant des constantes de région.

5.2.2. Justification des restrictions 1, 2, 3 et 4

Nous tenterons maintenant de justifier les restrictions imposées. Cette justification n'est pas toujours possible à ce niveau car beaucoup de restrictions ont été introduites à cause du traitement que l'on désire réaliser.

---

* Une instruction constante est une instruction non définie dans la région. Les instructions constantes sont, en règle générale, les instructions invariantes qui ont été déplacées. Une instruction constante portera également, dorénavant, le nom de constante de région.

1° On exige que la quantité I récursivement assignée soit une variable simple de type entier pour les raisons suivantes :

a) Des instructions du type I * K, où K est une constante de région, seront transformées en des additions successives.

<u>Exemple</u>

```
I = 1

L : D   N * I
    A [I] = B [J]
    I  =  I + 1
    IF (I.LE.100) GOTO L
```

sera transformé en

```
I = 1
t = N * I

L : J = t.
    A [J] = B [J]
    I := I + 1
    t := t + N
IF (I.LE.100) GOTO L
```

ou encore, sous une forme plus optimale

```
t = N
t1 = 100 * N

L : A [t] = B [t]
    t = t + N
    IF (t.LE.t1) GOTO L
```

Si I n'était pas un entier, comment **exprimer** par exemple 2.5 * N sous forme d'additions successives.

b) I pourrait être une variable indicée mais alors les indices doivent être des constantes de région. Généralement, on préfère exiger que I soit une variable simple, pour simplifier la tâche de l'optimiseur: ainsi il ne doit pas analyser les indices de la variable.

On refuse de considérer les variables indicées pour d'autres raisons que l'on mettra en évidence sur l'exemple suivant.

```
I = N
IV [I] = 1
L : J = K * IV [I]
    A [J] = B [J]
    I = I + 1
    IV [I] = IV [I] + 1
    IF (IV [I] .LE.100) GOTO L
```

devrait être transformé de la façon suivante

```
I = N
IV [I] = 1
t = K * IV [I]
L : J = t
    A [J] = B [J]
    I = I + 1
    IV [I] = IV [I] + 1
    t = t + K
    IF (IV [I] .LE.100) GOTO L
```

qui est un programme non équivalent au premier, puisque I varie dans la région.

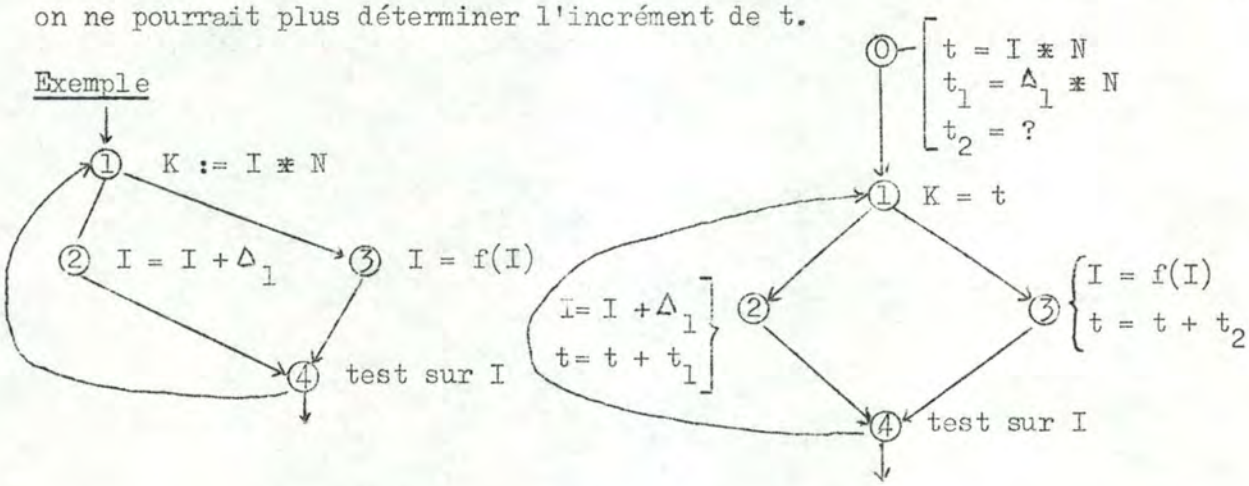
2° Le fait que V = f(V) appartient à une région nous permet de transformer certains opérateurs en une combinaison d'autres opérateurs.

3° Le fait que V = V + DELTA et non pas V = f(V) où f est une fonction quelconque nous est également imposé par le traitement que l'on désire réaliser.

En effet, si V = f(V) avec f quelconque, alors par exemple V * K (où K est une constante de région) n'est plus remplaçable par une addition.

L'exigence "DELTA est soit une constante de région, soit une instruction constante" provient du fait que l'on désire que l'incrément de la variable temporaire (t dans les exemples) soit calculable en dehors de la région. Si ce n'était pas le cas, nous détériorerions le programme au lieu de l'améliorer.

4° Si une des instructions modifiant la valeur de V dans la région n'était pas une assignation récursive au sens de 3°, le traitement que l'on désire réaliser ne conserverait plus l'équivalence des programmes. De plus, on ne pourrait plus déterminer l'incrément de t.



5.3. Opérateurs que l'on désire remplacer (réduire) et opérateur qu'il est impossible de réduire

5.3.1. Opérateur que l'on désire réduire

Nous décrirons ici le traitement que l'on réalisera sur des variables qui répondent aux critères précédents.

Certaines assignations dont les parties droites sont des fonctions de variables récursivement assignées sont elles-mêmes réductibles à des assignations récursives.

Soit I une variable récursivement assignée dans une région R et soient $I = I + D_1$, $I = I + D_2$, $I = I + D_3$, ..., $I = I + D_n$ les assignations récursives appartenant à R.

Une constante de région sera notée RC.

5.3.1.1. Réduction de I * RC ou RC * I

L'assignation t = I * RC ou t = RC * I peut être réduite à une assignation récursive par l'algorithme suivant, où t est une variable temporaire.

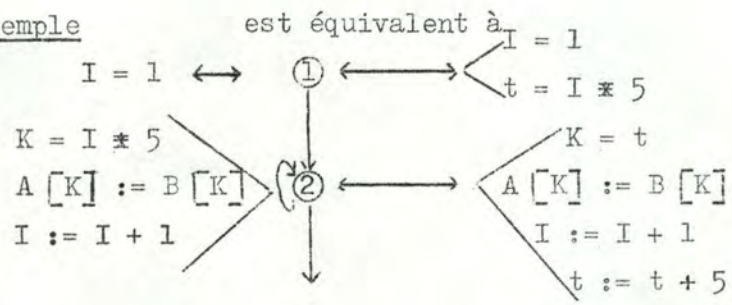
a) Placer t = I * RC (RC * I) devant chaque entrée de la région. Ceci a pour effet d'initialiser t de façon correcte.

b) Calculer DD_i = RC * D_i (D_i * RC). Ce calcul peut être fait directement si les valeurs de RC et de D_i sont connues. Sinon, on place DD_i = RC * D_i (D_i * RC) devant chaque entrée de la région R.

c) Placer t = t + DD_i après chaque assignation récursive $I = I + D_i$. DD_i est soit positif, soit négatif. Si $D_i > 0$ et $RC > 0$ alors $DD_i > 0$
Si $D_i < 0$ et $RC < 0$ alors $DD_i > 0$
Si D_i et RC sont de signes opposés alors $DD_i < 0$

Si, dans t = I * RC, RC n'était pas une constante de région, la transformation produirait une détérioration du programme puisque DD_i n'est plus une constante de région.

Exemple



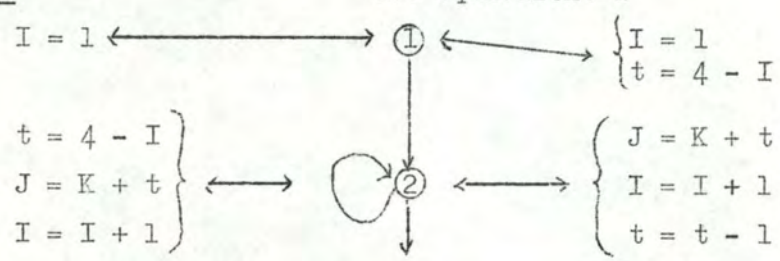
5.3.1.2. Réduction de I ± RC ou RC ± I

L'assignation t = I ± RC ou t = RC ± I peut être réduite à une assignation récursive par l'algorithme ci-dessous.

a) Placer t = I ± RC (t = RC ± I) devant toute entrée de la région R.

b) Placer $t = t + D_i$ ($t = t \pm D_i$) après chaque assignation récursive $I = I + D_i$.

<u>Exemple</u> est équivalent à



5.3.1.3. <u>Réduction de RC ** I</u>

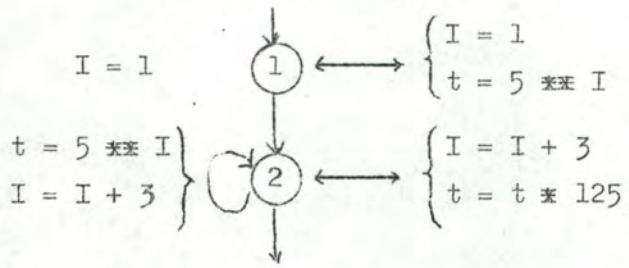
L'assignation t = RC ** I est réductible à une assignation récursive par l'algorithme suivant :

a) Placer t = RC ** I devant chaque entrée de R.

b) Calculer DD_i = RC ** D_i. Ce calcul peut être réalisé directement si les valeurs de RC et de D_i sont connues. Sinon on place DD_i = RC ** D_i devant chaque entrée de la région R.

c) Placer t = t * DD_i après chaque assignation récursive $I = I + D_i$.

<u>Exemple</u> est équivalent à



5.3.1.4. <u>Justification</u>

Nous pouvons démontrer l'exactitude algébrique de ces transformations. Cette exactitude algébrique disparaît si l'on traite autre chose que des entiers, par exemple des réels: dans ce cas, les erreurs d'arrondi ne sont plus les mêmes.

1) $t_{nouveau} = (I + D) * RC = I * RC + D * RC$

$t_{ancien} = I * RC$

$t_{nouveau} - t_{ancien} = D * RC$

ou $t = t + D * RC$

$$2)\ t_{nouveau} = (I + D) \pm RC = I + D \pm RC$$
$$t_{ancien} = \qquad\qquad = I \pm RC$$

$$t_{nouveau} - t_{ancien} = D$$
$$\text{ou } t = t + D$$

$$3)\ t_{nouveau} = RC \ast\ast (I + D) = (RC \ast\ast I) + (RC \ast\ast D)$$
$$= t_{ancien} \ast (RC \ast\ast D)$$
$$\text{ou } t = t \ast (RC \ast\ast D)$$

$$4)\ t_{nouveau} = RC \pm (I + D) = RC \pm I \pm D$$
$$t_{ancien} \qquad\qquad = RC \pm I$$

$$t_{nouveau} - t_{ancien} = \pm D$$
$$\text{ou } t = t \pm D$$

5.3.1.5. Remarques

$I \ast\ast RC$ n'est pas une opération réductible. On pourrait penser à utiliser la formule du "binôme de Newton" pour évaluer l'expression $(I + \Delta) \ast\ast RC$ mais la nouvelle forme de l'expression est fort peu maniable.

$$t_{nouveau} = (I + \Delta) \ast\ast RC = \sum_{i=o}^{RC} C_i\ I^{(RC-i)}\ \Delta^i$$
$$= t_{ancien} + \sum_{i=1}^{RC-1} C_i \frac{t_{ancien}}{t^i_{ancien}} \cdot \Delta^i + \Delta^{RC}$$

Les coefficients C_i sont fonction de RC qui est une constante de région mais il est impossible d'éliminer les divisions $t_{ancien}/t_{ancien} \ast\ast i$.

5.3.2. Opérateur qu'il est impossible de réduire sans imposer des conditions supplémentaires

Cet opérateur est l'opérateur de division entière. En effet, soient l'expression I/RC et la variable I définie récursivement par les assignations récursives $I := I + D_1, \ldots, I := I + D_n$.

$$t_{nouveau} = (I + D_i)/RC = I/RC + D_i/RC$$
$$= t_{ancien} + D_i/RC$$

Cette expression est correcte algébriquement mais ne l'est plus en machine. On pourrait penser à la transformation suivante comme transformation découlant du calcul précédent :

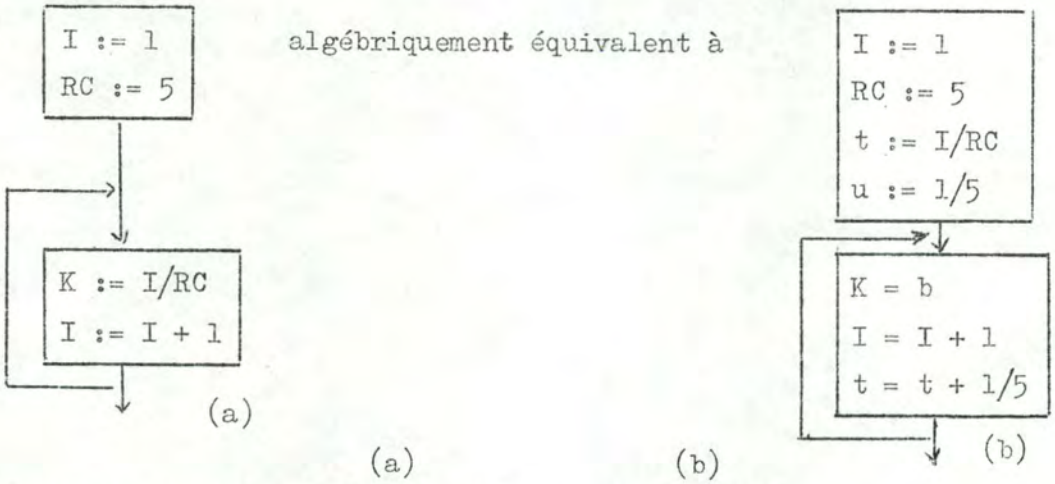
a) Placer $t = I/RC$ devant chaque entrée dans la région.

b) Calculer $u_i = D_i/RC$ directement si D_i et RC sont des constantes. Sinon placer $u_i = D_i/RC$ devant chaque entrée dans la région.

c) Placer $t = t + u_i$ après chaque instruction $I = I + D_i$.

Nous montrerons maintenant sur des exemples que cette transformation n'est pas correcte.

1)



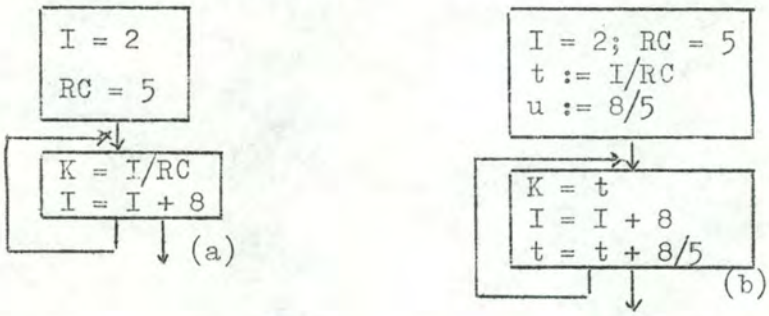
	(a)	(b)
1er passage dans R	K := 0, I = 2	K = 0, I = 2, t = 0
2me passage dans R	K := 0, I = 3	K = 0, I = 3, t = 0
3me passage dans R	K := 0, I = 4	K = 0, I = 4, t = 0
4me passage dans R	K := 0, I = 5	K = 0, I = 5, t = 0
5me passage dans R	K := 1, I = 6	K = 0, I = 6, t = 0
6me passage dans R	K := 1, I = 7	K = 0, I = 7, t = 0

⋮

On remarque donc que dans le programme transformé, K reste égal à zéro, tandis que dans (a), K passe à la valeur 1.
Cet exemple met en doute l'équivalence de la transformation.

Un autre exemple va montrer que la transformation met en doute "l'équivalence sémantique".

2)



	(a)	(b)
1er passage	K = 0, I = 10	K = 0, I = 10, t = 1
2me passage	K = 2, I = 18	K = 1, I = 18, t = 2
3me passage	K = 3, I = 26	K = 2, I = 26, t = 3

Les raisons de cette non équivalence semblent provenir des causes suivantes :

1° Dans l'exemple 1: D était strictement inférieur à RC.

2° Dans l'exemple 2: $I_{initial} = Q_{I_i} * RC + R_{I_i}$ et $D = Q_D * RC + R_D$ et $R_{I_i} + R_D$ est un multiple de RC.

En règle générale, nous pouvons donc affirmer que la transformation n'est pas correcte pour le cas de la division.

Pour éviter les divisions dans une région, certains auteurs (Allen, Medlock) proposent de transformer une instruction du type A/B en u = 1/B, A * u, la division 1/B serait calculée à l'extérieur de la région et avant d'entrer dans la région. Pour que cela soit possible, il faut que 1/B puisse être déplaçable en arrière. Cette transformation bien qu'algébriquement correcte ne l'est généralement plus si l'on tient compte des réalités matérielles (exemple 1). Ceci est vrai pour les divisions entières, pour les divisions de type réel la transformation peut donner des résultats valables.

5.4. Introduction des notions d'instruction réductible et de séquence d'instructions réductibles

5.4.1. Définition d'une instruction réductible

Une instruction réductible est une opération ayant pour opérandes une constante de région et une variable récursivement assignée ou une instruction réductible.

Plus précisément, les instructions réductibles sont les suivantes (I est une variable récursivement assignée et RC une constante de région) :

1°) les opérations de base

(1) I * RC, (2) RC * I, (3) I + RC, (4) RC + I, (5) RC - I, (6) I - RC, (7) RC ** I

2°) les opérations du type

(11) α_i * RC, (22) RC* α_i, (33) α_i + RC, (44) RC + α_i, (55) RC - α_i, (66) α_i - RC, où α_i est une des instructions (1) à (6) ou (11) à (66).

3°) l'opération RC ** α_i où α_i est une des instructions (1) à (6) ou (11) à (66).

Toute autre instruction est dite irréductible.

Remarquons que toute utilisation du résultat de l'opération réductible (7) au (3°) devient une opération irréductible.

5.4.2. Définition de séquence d'instructions réductibles ou séquence réductible

Une séquence d'instructions I_1, I_2, ..., I_n appartenant à un même segment est dite être une séquence "d'instructions réductibles",

si et seulement si

1° chacune d'elles est réductible;

2° I_i utilise le résultat I_{i-1}.

Exemples

1) α_1 I * RC1
α_2 α_1 + RC2
α_3 α_2 * RC3
α_4 RC4 - α_3

RC4 - ((I * RC1 + RC2) * RC3)

2) α_1 RC1 + RC2
α_2 α_1 + I
α_3 RC3 + α_2

$\longrightarrow$ séquence réductible

3) α_1 A * I
α_2 C * D
α_3 A * B
α_4 α_1 + α_2
α_5 α_4 + α_3

- α_1, α_4, α_5 est une séquence d'instructions réductibles
- A, B, C, D sont des R.C.

L'exemple 3 montre qu'il est utile que le déplacement d'instructions se déroule avant la réduction des opérateurs, pour les raisons suivantes:

1° une plus grande facilité dans la détection des instructions constantes;

2° l'instruction α_2 est invariante dans la région et déplaçable en arrière (voir critère de déplacement en arrière). Si ce déplacement en arrière

vers le segment fictif n'avait pas eu lieu, la réduction de α_4 placerait dans ce segment une instruction dont un opérande (α_2) ne serait pas défini dans le segment fictif.

Une séquence d'instructions réductibles se termine dès qu'une instruction non réductible est trouvée. Ces instructions non réductibles qui suivent la séquence peuvent être des utilisations d'instructions réductibles figurant dans la séquence, auquel cas elles terminent la séquence.

α_1 I * RC_1
α_2 α_1 + RC_2
α_3 α_3 + RC_3
α_4 J := α_1
α_5 K := α_2
α_6 M := α_3
α_7 α_3 * RC_4
α_8 L := α_7

J := α_1 termine la séquence réductible α_1
K := α_2 termine la séquence réductible α_1, α_2
M := α_3 termine la séquence réductible α_1, α_2, α_3
L := α_7 termine la séquence réductible α_1, α_2, α_3, α_7

Pour plus de clarté, nous donnons une définition constructive d'une séquence d'instructions réductibles.

a) Une instruction du type (1) à (6) débute une séquence d'instructions réductibles.

b) Une instruction du type (11) à (66) complète une séquence d'instructions réductibles si elle utilise le résultat de la dernière instruction réductible figurant dans la séquence.

c) Toute autre instruction termine une séquence d'instructions réductibles. Elle peut la terminer de trois façons différentes :

1° l'instruction utilise le résultat d'une instruction figurant dans la séquence réductible;

2° l'instruction n'utilise le résultat d'aucune instruction figurant dans la séquence réductible;

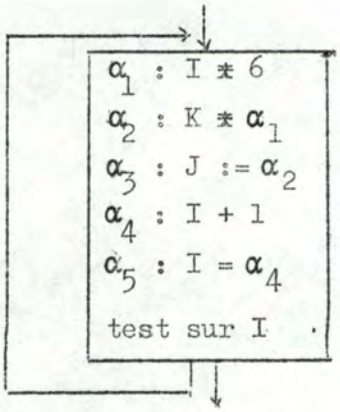
3° l'instruction est du type (7) ou 3°.

5.5. Utilisation de la notion de séquence réductible

Lorsque nous sommes en présence d'une assignation dont la partie droite est une expression réductible (exemple K = I * RC), une variable est presque toujours générée par l'optimiseur pour exprimer l'assignation récursive qui résulte de la réduction, ceci à cause de l'analyse abondante requise pour déterminer si la variable assignée (K ici) est déplaçable en dehors de la région.

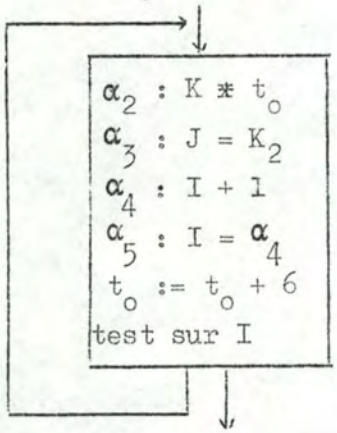
Le remplacement de chaque instruction réductible par une assignation récursive implique que les variables générées t sont elles-mêmes récursivement assignées dans la région et par conséquent les utilisations de ces variables deviennent des instructions candidates à une réduction. Ces réductions successives introduiront de nouvelles variables générées et de plus en plus d'assignations récursives. L'utilisation de la notion de séquence réductible permettra en grande partie d'éviter ces générations de variables t et l'introduction d'assignations récursives.

Exemple



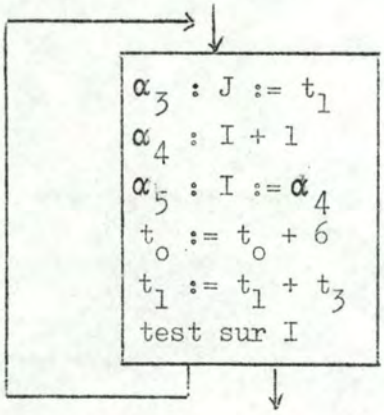
1) Réduction de I * 6

t_o = I * 6



2) Réduction de $K * t_o$

$t_o = I * 6$
$t_1 = K * t_o$
$t_3 = K * 6$



Remarques sur l'exemple

- t_o, après la réduction de $K * t_o$, n'est plus référencé dans la région. Ceci est généralement vrai si toutes les utilisations de t_o sont réductibles.

- Il est par conséquent inutile de conserver dans la région l'assignation récursive $t_o := t_o + 6$.

Pour éviter l'introduction de variables inutilisées et dès lors d'assignations récursives non nécessaires, des séquences d'instructions réductibles sont constituées et remplacées par une seule variable t récursivement assignée.

La procédure qui réalisera cette opération simule les introductions de variables générées et n'introduit effectivement de variables générées que lorsqu'elle rencontre une instruction non réductible utilisant le résultat d'une instruction réductible figurant dans la séquence. Pour ce faire, la procédure disposera d'une table DELTATABLE qui contiendra les différents incréments des variables générées simulées et effectivement générées.

Si nous appliquons la procédure à l'exemple précédent, nous obtenons le résultat suivant :

DELTATABLE I

α_1	6	α_2	K*6	

α_1 : I * 6
α_2 : K * α_1
α_3 : t = α_2
α_4 : K * 6
α_5 : t_1 = α_4

J := t
I := I + 1
t *= t + t_1

Ainsi la séquence

α_1 : I * 6, α_2 : K * α_1 a été réduite à t := t + t_1

Le résultat d'une instruction réductible peut être utilisé par plusieurs instructions. Certaines sont réductibles et, par conséquent, seront introduites dans la séquence; d'autres ne sont pas réductibles et terminent donc la séquence.

Exemple

α_1 : I * 5
α_2 : J := α_1
α_3 : α_1 * 4
α_4 : K := α_3
α_5 : M := α_3
α_6 : α_3 + JJ
α_7 : L := α_6
α_8 : α_6 + LK
α_9 : LL := α_8
α_{10} : I + 1
α_{11} : I := α_{10}
test sur I

est équivalent à

t_1 := I * 5
t_2 := t_1 * 4
t_3 := t_2 + JJ
t_4 := t_3 + LK

α_2 : J := t_1
α_4 : K := t_2
α_5 : M := t_2
α_7 : L := t_3
α_9 : LL := t_4
α_{10} : I + 1
α_{11} : I := α_{10}
t_1 := t_1 + 5
t_2 := t_2 + 20
t_3 := t_3 + 20
t_4 := t_4 + 20
test sur I

α_2 termine la séquence $\{\alpha_1\}$
α_4 et α_5 terminent la séquence $\{\alpha_1, \alpha_3\}$
α_7 termine $\{\alpha_1, \alpha_3, \alpha_6\}$
α_9 termine $\{\alpha_1, \alpha_3, \alpha_6, \alpha_8\}$

Chaque fois qu'une instruction non réductible termine une séquence, il faut se fournir une variable pour exprimer la séquence réductible sous forme d'une assignation.

Les règles suivantes sont valables pour déterminer la variable qui exprimera la séquence.

1° Si l'instruction non réductible qui termine la séquence $s \equiv I_1, \ldots, I_n$ n'est pas la première à avoir terminé la même séquence s, alors le nom généré pour les terminaisons précédentes de s est utilisé. Comme cette variable générée a déjà été initialisée et incrémentée, ceci n'est pas refait;

2° Si l'instruction non réductible ne satisfait pas à la condition 1° ci-dessus, alors un nom de variable est généré.

<u>Exemple</u> Dans l'exemple précédent, α_5 : M := α_3 termine la séquence $\{\alpha_1, \alpha_3\}$ sous la condition 1°. Tandis que toutes les autres instructions α_2, α_4, α_7, α_9 terminent des séquences sous la condition 2°.

5.6. Avantage de la notion de séquence réductible et de la réduction d'opérateurs à des opérateurs plus efficients *

1 Si le programme objet est exploité sur une machine qui possède des registres d'index, alors les opérations réduites peuvent devenir des incrémentations ou des décrémentations de registres d'index. Ceci est également une justification de la condition 1° du paragraphe 4.2.1.

2° L'utilisation de la notion de séquence d'instructions réductibles permet de réduire plusieurs opérations à une seule assignation récursive.

3° Le nombre d'utilisations de la variable I récursivement assignée est diminué, peut-être même annulé. Ceci nous permettra peut-être d'éliminer du programme les assignations récursives I := I + D_i. Cette procédure sera décrite dans le chapitre VI.

---

* Un opérateur plus efficient est un opérateur tel qu'il lui correspond des instructions machine moins coûteuses en temps.

Ce qui est présenté dans ce chapitre n'est qu'une simple extension de ce que les compilateurs peu optimisants réalisent pour des boucles DO (FORTRAN IV) ou for (ALGOL 60).

5.7. Procédure de réduction des opérateurs

5.7.1. Remarques préliminaires

5.7.1.1. La réduction des opérateurs pour une certaine région R_i est exécutée après le déplacement d'instructions invariantes depuis R_i mais avant l'insertion du segment fictif, car celui-ci devra contenir les instructions d'initialisation.

Juste avant la réduction des opérateurs, le programme en forme intermédiaire possède les propriétés suivantes :

5.7.1.1.1. la liste R_i' contenant les segments de R_i qui n'ont pas encore été traités est connue;

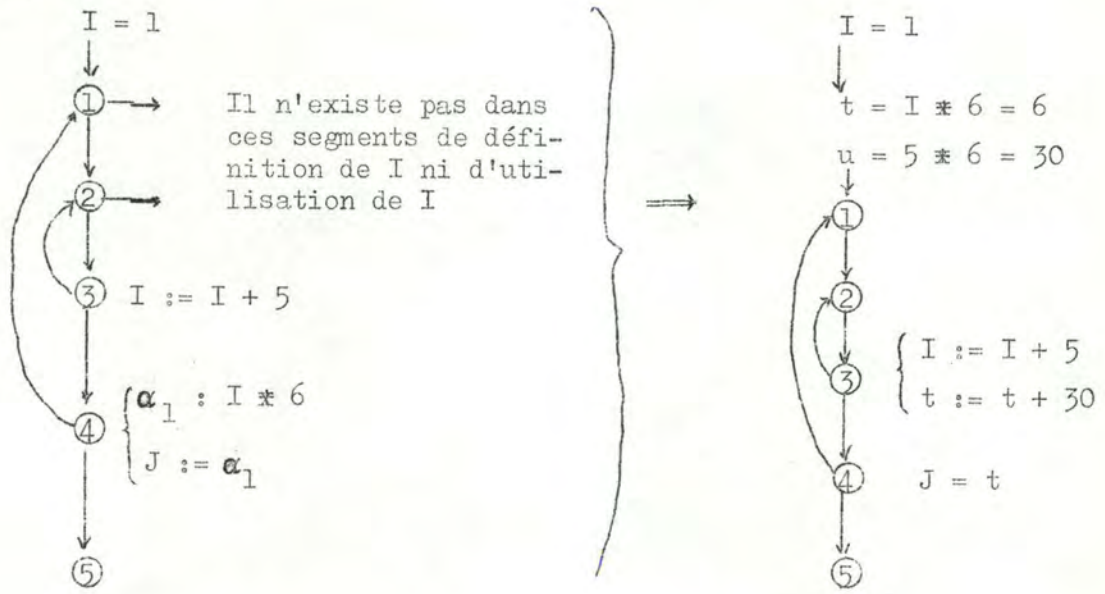
5.7.1.1.2. les tables ASS et REF sont connues en ce sens que les vecteurs-lignes ASS [variable V] et REF [variable V] sont accessibles pour chaque variable au tableau;

5.7.1.1.3. s'il existe dans R_i' des instructions déplaçables, celles-ci se trouvent dans le segment fictif, ce qui signifie que certaines instructions intermédiaires utilisent des instructions intermédiaires qui ne sont pas définies dans le segment. Ces instructions intermédiaires utilisées mais non définies dans le segment peuvent être considérées comme des R.C.

5.7.1.2. Il ne faut considérer que R_i' pour trouver les instructions réductibles. En effet, si une instruction réductible OP; I; RC se trouve dans une région interne R_j de R_i et si I est une variable récursivement assignée dans R_j, alors OP; I; RC a été réduite précédemment. Sinon, soit elle a été déplacée en arrière de la région interne, soit elle n'est pas réductible dans R_j et donc, a fortiori, dans R_i.

5.7.1.3. La recherche des assignations récursives se fait dans l'ensemble de la région R_i, pour la raison suivante: si $I := I + D_i$ se trouve dans R_j (région interne de R_i), $I := I + D_i$ ne pourra pas être déplacé ni en avant, ni en arrière. Par conséquent, il existe dans la région interne une instruction modifiant récursivement la valeur de I.

Exemple



Si nous n'examinions pas l'ensemble de la région $\{1, 2, 3, 4\}$ pour la localisation des assignations récursives, l'opération I * 6 ne serait pas réductible, ce qui serait fâcheux.

5.7.1.4. Les instructions réductibles dans une région R_i et telles que la variable de récursion I est assignée récursivement dans R_i ne sont pas déplaçables en dehors de R_i.

5.7.1.5. La réduction des opérateurs se passe après l'élimination d'instructions redondantes pour faciliter la localisation d'instructions réductibles.

5.7.1.6. (5.7.1.4) nous permet d'affirmer que, si le résultat d'une instruction réductible I située dans un segment est utilisé dans ce segment, l'utilisation se trouve dans le segment après I. Par conséquent, la recherche de séquences réductibles peut se réaliser segment par segment.

5.7.1.7. Si I := I + α_i est une assignation récursive, alors α_i est une instruction constante dans la région. De plus, si α_i dépendait du résultat d'autres instructions intermédiaires, celles-ci seraient également des instructions constantes. Par conséquent, l'instruction intermédiaire α_i et toutes les instructions intermédiaires dont α_i dépend sont déplacées en dehors de R_i.

5.1.7.8. Si $I_1, I_2, \ldots, I_n$ est une séquence d'instructions réductibles, alors, pour tous i et j tels que $i < j$ et $i,j \in \{1, 2, \ldots, n\}$: $n(I_i) < n(I_j)$.

Plus précisément, si I_j utilise le résultat de I_i, alors $n(I_j) = n(I_i)+1$. En effet, $n(I_j) = \max(n(I_i), n(RC)) + 1 = n(I_i) + 1$ (vu que $n(RC) = 0$).

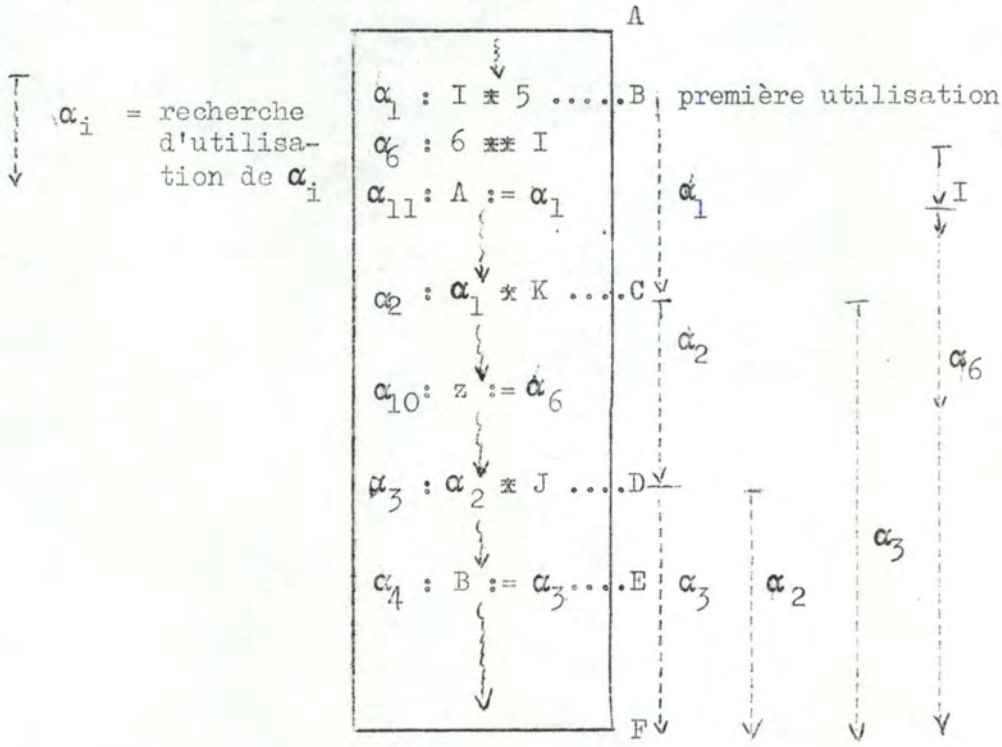
Ceci signifie que, si I_i est utilisé dans une instruction réductible, celle-ci figure après I_i et parmi les instructions ayant un numéro de niveau égal à $n(I_i) + 1$.

5.7.2. Procédure de réduction des opérateurs

Nous exposerons les idées de base de la procédure qui s'appuient sur l'ensemble des paragraphes précédents. Nous mettrons en évidence, grâce à un exemple, la façon dont la procédure traite un segment S.

5.7.2.1. Exemple

Soit un segment S et une variable récursivement assignée I définie par l'assignation I := I + D.



La procédure commencera par scanner le segment S jusqu'à trouver la première utilisation réductible de la variable récursivement assignée I. Dès que la procédure trouve cette instruction, elle la réduit.

Dans notre exemple, la première instruction réductible utilisant I est α_1: I * 5. Les actions suivantes sont exécutées :

1°) α_1 : I * 5 est notée comme ne devant pas être généré lors de la génération du code objet.

2°) α_1 : I * 5 est placée dans le segment fictif où elle est notée comme devant être générée.

3°) Si D est une constante, l'incrément D * 5 est calculé directement et sa valeur est placée dans la Deltatable. Si D n'est pas une constante, l'expression D * 5 est placée dans la Deltatable. La Deltatable se présente alors de la façon suivante :

D	α_1	D * 5	.....

La procédure continue ensuite à scanner le segment afin d'y rechercher les utilisations de α_1. Ces utilisations peuvent être réductibles ou non réductibles.

La première utilisation rencontrée est α_{11} : A := α_1 qui est non réductible. Les actions suivantes sont exécutées :

1°) t := α_1 est inséré dans le segment fictif où t est une variable générée selon les critères du paragraphe 4.5.

2°) Après chaque assignation récursive I := I + D - si D est une constante, l'instruction t := t + 5 * D est insérée; 5 * D est la valeur se trouvant dans la Deltatable.

- si D n'est pas une constante, l'instruction t := t + u est insérée, où u est une variable générée et initialisée à 5 * D dans le segment fictif.

3°) L'instruction A := α_1 est remplacée par A := t.

La procédure continue ensuite son analyse du segment pour trouver des utilisations de α_1. Elle trouve α_2 : α_1 * K qui est une utilisation réductible de α_1. La procédure produit la réduction comme pour α_1, ce qui a pour effet de modifier la Deltatable qui devient

D	α_1	D * 5	α_2	D*5*K	...

, et de déposer l'instruction α_2 : α_1 * K dans le segment fictif.

Ensuite la procédure sconne le segment pour trouver des utilisations de α_2 parmi lesquelles certaines sont réductibles et d'autres non. Ces utilisations sont traitées d'une façon identique à celle de α_1. Soit α_3: $\alpha_2 * J$ la première utilisation réductible de α_2; on réalise la réduction qui forme une nouvelle Deltatable

D	α_1	D*5	α_2	D*5*K	α_3	D*5*K*J	...

.

Ensuite, la procédure recherche lesutilisations de α_3. Soit α_4 l'utilisation de α_3, elle est non réductible. On traite ce cas de non réduction comme précédemment.

Puis la procédure scanne le segment jusqu'à la fin pour trouver les utilisations de α_3. Lorsque toutes ont été traitées,

α_3	D*5*K*J

est enlevé du sommet de la table Deltatable. On recommence à chercher les utilisations de α_2 à partir de l'instruction qui suit α_3 dans S. On traite suivant le cas toutes les utilisations de α_2. Lorsqu'elles ont toutes été traitées, on enlève du sommet de la Deltatable

α_2	D*5*K

.

On recherche ensuite toutes les utilisations de α_1, on les traite et on supprime de la Deltatable

α_1	D*5

.

Dès ce moment, on peut sconner le segment à partir de l'instruction qui suit α_1, pour trouver d'autres utilisations réductibles de I et les utilisations de ces résultats.

Lorsque toutes les utilisations réductibles de I ont été traitées, on passe à un autre segment S' et on agit comme précédemment. Si tous les segments appartenant à la région réduite R' ont été traités, on recommence le tout pour une autre variable récursivement assignée. La procédure se termine lorsque toutes les variables récursivement assignées dans R ont été traitées.

<u>Remarques</u> 1) Deltatable contiendra toujours la séquence d'instructions réductibles en cours de construction.

2) Deltatable se comporte comme une pile dont le sommet contient toujours l'instruction dont on cherche les utilisations.

Lorsque toutes les utilisations de cette instruction ont été traitées, on enlève du sommet de la pile l'instruction et son incrément. Et on recommence le processus pour la nouvelle instruction au sommet de la pile et à partir de l'instruction suivant l'ancienne instruction au sommet de la pile.

5.7.2.2. Procédure de réduction des opérateurs

1) A l'aide des tables ASS et REF, localisons dans R_i les assignations récursives au sens de 4.2.1.

Lister ces assignations en les groupant par variable récursivement assignée, avec leurs incréments et leur position dans la région R_i. Soit LASR la liste.

2) Créer, à partir de la liste précédente, une liste de variables récursivement assignées au sens de 4.2.1.

Soit RAV = $\{ V_1, V_2, \ldots, V_m \}$ cette liste.

3) $i := 1$

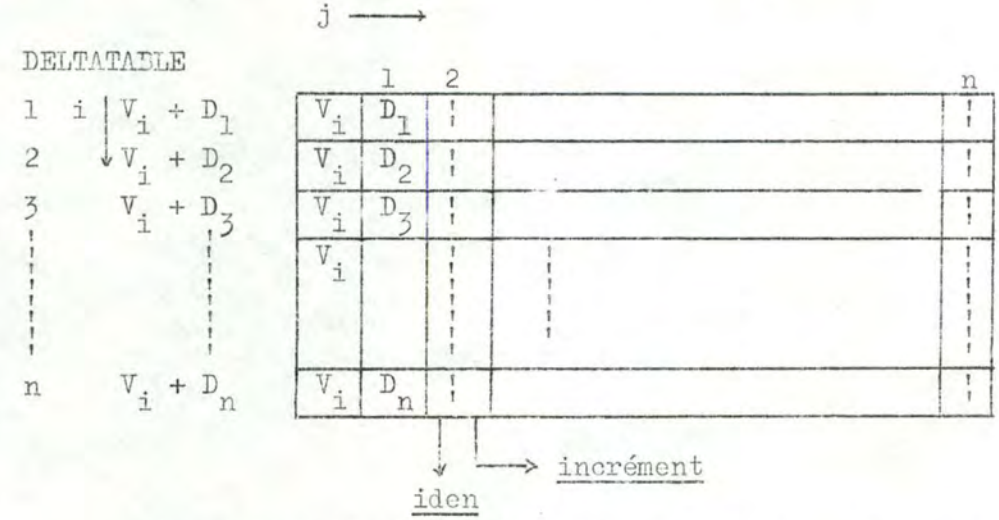
4) Sélectionner V_i dans RAV et ses assignations récursives dans LASR, soient $V_i := V_i + D_1$; ....., $V_i := V_i + D_n$.

5) Créer une table "DELTATABLE" à deux dimensions pour V_i.

Dans la table, il y a une entrée (qui est une pile) par assignation récursive $V_i := V_i + D_j$, donc n piles.

Initialiser la pile Deltatable de la façon suivante :

Deltatable $[1,1] := D_1$, Deltatable $[2,1] := D_2$, ..., Deltatable $[n,1] := D_n$.



Iden.deltatable $[i,j]$ ($i := 1, \ldots, n$ et $j > 1$) contiendra l'identificateur de l'instruction réductible pour laquelle Increment.deltatable $[i,j]$ contient l'incrément cumulé résultant de la réduction de la séquence des j-1 instructions réductibles figurant dans la deltatable.

6) Soit $R'_k = S_1, S_2, \ldots, S_e$ l'ensemble des segments non encore optimisés par la réduction des opérateurs dans la région R_k.

7) ii := 1;

8) Prendre S_{ii} dans R'_k. Scanner les instructions de S_{ii} afin d'y rechercher les instructions réductibles pour la variable V_i choisies en (4) et les instructions non réductibles utilisant le résultat d'instructions réductibles.

(8).(1) Si l'on trouve une instruction réductible I autre que l'exponentiation et qui complète la séquence en cours, les actions suivantes sont exécutées.

Soit (Deltatable $[i,n]$ | n fixé, $(i = 1,\ldots, n)$ l'ensemble des dernières zones garnies dans la table, et soit Iden.deltatable $[i,n] \underset{\text{notation}}{=} \alpha_{in}$ et incrément.deltatable $[i,n] \underset{\text{notation}}{=} D_{in}$ pour $(i=1,\ldots,n)$. Il est à noter que $\alpha_{1n} = \alpha_{2n} = \ldots = \alpha_{nn}$.

(8).(1).(0) Noter, dans S_{ii}, I comme ne devant pas être générée.

(8).(1).(1) L'instruction est placée dans le segment fictif après toutes celles qui y figurent déjà.

(8).(1).(2) Calculer les incréments cumulés et placer les résultats dans Increment.deltatable $[i,n+1]$ *.

De même, on place dans Iden.deltatable $[i,n+1]$ l'identificateur de l'instruction I qui a donné lieu à la réduction.

(8).(1).(3) Calcul des incréments cumulés $(ARG = \alpha_{in})$

(8).(1).(3).(1) Si I est du type ARG * RC alors l'incrément sera D_{in} * RC $\forall$ i.

(8).(1).(3).(2) Si I est du type RC * ARG alors l'incrément sera RC * D_{in} = D_{in} * RC $\forall$ i.

(8).(1).(3).(3) Si I est du type RC + ARG alors l'incrément sera D_{in} $\forall$ i.

(8).(1).(3).(4) Si I est du type RC - ARG alors l'incrément sera - D_{in} $\forall$ i.

---

* Si l'incrément n'est pas une opération impliquant des constantes, on place dans Incrément.deltatable l'expression qui permet de calculer la valeur de l'incrément.

(8).(1).(3).(5) Si I est du type ARG + RC alors l'incrément sera D_{in} ∀ i.

(8).(1).(3).(6) Si I est du type ARG - RC alors l'incrément sera D_{in} ∀ i.

(8).(1).(4) Continuer à scanner S_{ii} pour rechercher les utilisations de l'instruction au sommet de la pile Deltatable.

(8).(2) Si l'on trouve une instruction réductible du type exponentiation α_k: RC ** α_{in}, alors les actions suivantes sont exécutées.

(8).(2).(4) L'instruction est notée comme ne devant pas être générée lors de la génération de code.

(8).(2).(2) Calculer le facteur multiplicatif pour RC ** α_{in}, c'est-à-dire RC ** D_{in}. Mémoriser ce facteur dans une zone de travail avec l'identificateur de l'instruction.

(8).(2).(3) Placer l'instruction α_k: RC ** α_{in} dans le segment fictif.

(8).(2).(4) Une instruction d'exponentiation qui est réductible termine une séquence d'instructions réductibles. Ceci est dû au fait que son utilisation par une autre instruction ne délivre jamais une instruction réductible.

Par conséquent, il faut générer une variable temporaire pour exprimer la réduction. A cette fin, nous appliquons les critères du paragraphe 4.5.

Si t_e est la variable temporaire choisie ou générée, alors nous plaçons dans le segment fictif l'instruction $t_e := \alpha_k$ et nous mémorisons dans une zone de travail que toute référence à α_k doit être remplacée par une référence à t_e.

Après chaque assignation récursive dans R_i $(V_i + D_j)$, nous plaçons $t_e := t_e * (RC ** D_{in})$.

Si RC est une constante, de même que D_{in}, alors RC ** D_{in} est calculable directement et $t_o = t_e * (RC ** D_{in})$ est effectivement inséré.

Si, par contre, RC ** D_{in} n'est pas calculable directement, l'instruction $t_o := t_e * u_i$, où u_i est initialisé à RC ** D_{in} dans le segment fictif, est insérée après $I := I + D_i$.

Dans la suite du segment, toute référence à α_k est remplacée par une référence à t_e.

Nous scannons ensuite S pour trouver les utilisations de α_{in}.

(8).(3) Lorsque nous trouvons une instruction non réductible n'utilisant pas α_{in}, aucune action n'est exécutée et nous poursuivons l'analyse S à la recherche d'utilisations de α_{in}.

(8).(4) Lorsque nous trouvons une instruction non réductible I utilisant le résultat de l'instruction α_{in}, celle-ci termine la séquence d'instructions réductibles en cours.

Les actions suivantes sont exécutées :

(8).(4).(1) Une variable temporaire t est générée selon les critères du paragraphe 4.5.

(8).(4).(2) Si D_{in} est calculable à la compilation, alors nous insérons après chaque assignation récursive $V_i := V_i + D_j$ l'assignation récursive $t := t + D_{jn}$.

Si D_{in} n'est pas une constante, nous insérons dans le segment fictif le code nécessaire au calcul de D_{in} et générons des variables temporaires u_i qui seront initialisées par le résultat de ce calcul. Cette initialisation se fait dans le segment fictif. Ensuite, nous plaçons, après chaque assignation $V_i := V_i + D_j$, l'assignation récursive $t := t + u_j$.

(8).(4).(3) Nous remplaçons la référence à α_{in} par une référence à t dans I.

(8).(4).(5) Ensuite, nous continuons à analyser le segment S pour rechercher d'autres instructions réductibles ou non utilisant α_{in}.

(8).(5) Lorsque dans S_{ii} toutes les instructions utilisant α_{in} ont été traitées, nous diminuons la pile deltatable d'un élément, c'est-à-dire nous enlevons l'élément $\boxed{\alpha_{in} \mid D_{in}}$ du sommet de la deltatable.

Ensuite, à partir de l'instruction suivant α_{in} dans S, nous analysons le reste de S pour rechercher les utilisations de α_{in-1} qui se trouve maintenant au sommet de la pile deltatable.

(8).(6) Lorsque toutes les utilisations des instructions se trouvant dans la pile, c'est-à-dire α_{i2}, α_{i3}, ..., $_{in-1}$ ont été traitées, la pile deltatable est prête pour que l'on envisage, à partir de l'instruction qui suit α_{i2} dans S, les nouvelles utilisations réductibles de la variable V_i.

9) Lorsque, pour V_i, toutes les utilisations réductibles et les utilisations des résultats de ces instructions appartenant à S_{ii} ont été traitées, alors nous faisons ii := i + 1 et nous recommençons le processus décrit par (8).

10) Si R'_k est épuisé, alors nous passons à la variable V_{i+1} et recommençons le processus décrit par (4), (5), (6), (7), (8), (9) et (10).

11) Si la liste RAV est épuisée, alors l'algorithme se termine.

5.8. Discussion sur la "réduction d'opérateurs à des opérateurs plus efficients

1) Il est important que la réduction des opérateurs se passe après le déplacement d'instructions invariantes.

Prenons l'exemple suivant pour appuyer notre discussion.

α_1(+; x; y)	(1) x et y sont des RC
α_2(*; α_1, I)	(2) I := I + D

Si nous n'avions pas déplacé les instructions invariantes, l'instruction α_2 apparaîtrait comme étant non réductible car α_1 est défini dans le segment.

2) Pour des additions de variables simples ou de constantes, il est intéressant d'ordonner les opérandes dans un certain ordre afin de permettre la réduction de certaines additions. La même remarque est valable pour des multiplications.

L'ordre pourrait, par exemple, être le suivant :

(1) On place comme premier opérande la variable récursivement assignée.

(2) Ensuite, les variables qui sont des constantes de région, en ordre lexicographique.

(3) Ensuite celles qui ne sont pas des constantes de région, en ordre lexicographique.

Exemple

Soit $A * C * I * B * D$ où I est la variable récursivement assignée, A et D des constantes de région, B et C des variables quelconques qui ne sont pas des constantes de région.

$\alpha_1(A * C)$	$\alpha'_1(I * A)$
$\alpha_2(\alpha_1 * I)$	$\alpha'_2(\alpha'_1 * D)$
$\alpha_3(\alpha_2 * B)$	$\alpha'_3(\alpha'_2 * B)$
$\alpha_4(\alpha_3 * D)$	$\alpha'_4(\alpha'_3 * D)$
(a)	(b)
forme intermédiaire non ordonnée	forme intermédiaire ordonnée

Dans (a), aucune instruction n'est réductible, alors que dans (b), α'_1 et α'_2 sont réductibles.

3) Pour permettre plus facilement la réduction d'opérateurs, il est intéressant que la partie variable du calcul d'adresse d'un élément d'un tableau à n dimensions soit exprimée comme une somme de n termes $(i_1 * d_2 * d_3 \ldots * d_n + i_2 * d_3 * d_4 * \ldots * d_n + \ldots + i_{n-1} * d_n + i_n$ où $d_i = u_i - e_{i+1}$ et u_i est la borne supérieure, e_i la borne inférieure de la $i^{\text{ème}}$ dimension) au lieu d'être exprimé sous la forme

$$((\ldots(((i_1 * d_2) + i_2) * d_3) + \ldots) + i_{n-1}) * d_n + i_n.$$

En effet, cette dernière ne permet que la réduction de $i_1 * d_2$ et pas celle de $(i_1 * d_2) + i_2$ ni des autres, tandis que l'autre forme permet la réduction de tous les termes de la somme. (Nous avons supposé que i_1, i_2, .., i_n étaient récursivement assignés).

4) Remarque sur la procédure: une table de nom généré lors de la réduction des opérateurs est tenue. Ceci permet, lorsque l'on est en présence d'une utilisation non réductible d'une instruction réductible, de remplacer la référence à l'instruction réductible par une référence à la variable générée correspondante qui exprime la réduction.

5) Il faut faire attention à la structure de bloc dans certain langage. Nous renvoyons le lecteur au chapitre IV pour les remarques concernant le déplacement d'instructions dans des langages à structure de bloc.

5.9. Optimisation particulière pour les boucles du type DO (FORTRAN IV) ou for (ALGOL 60) transformée et statique

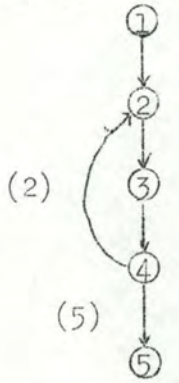
5.9.1. Définition de la notion "variable réinitialisable"

Une variable t générée lors de la réduction des opérateurs dans une région R est une variable réinitialisable si son évolution totale est connue à la sortie de la région R, lors de la compilation.

Par évolution totale de la variable, nous entendons le nombre n de fois que l'on a incrémenté la variable t par une assignation **t** := t + D lors de l'exécution répétitive de R.

Lorsque l'on connaît cette valeur n, on peut remettre t à la valeur qu'il possédait avant d'entrer dans R par l'insertion après la sortie de R de l'instruction t := t - (n * D).

Exemple



t := 1

t := t + 1

if t ≤ 100 then (2) else (5)

A la sortie de R ≡ { 2, 3, 4 } , t aura la valeur 101. Son évolution totale est connue: il aura été incrémenté 100 fois de 1 unité.

5.9.2. Utilisation des variables réinitialisables

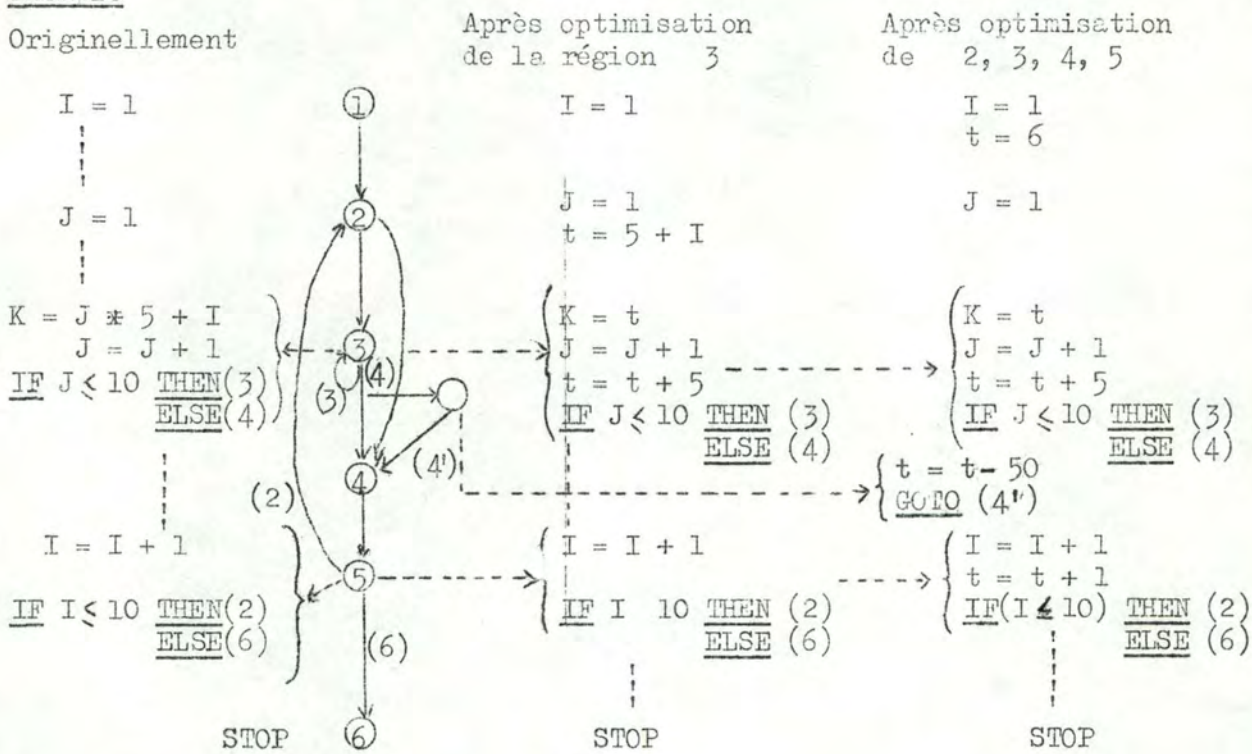
L'idée poursuivie est de limiter le nombre de variables temporaires générées, en réutilisant pour la réduction des opérateurs, celles qui sont générées pour une région interne qui satisfait à certains critères.

Soient R_i et R_j deux régions telles que R_j couvre R_i. Supposons qu'une variable t est générée lors de la réduction d'instructions de R_i.
Dans R_j, juste avant l'entrée dans R_i, on trouve les instructions qui initialisent t à la valeur requise.

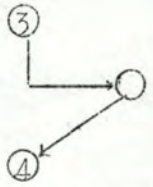
Si tout ou partie des instructions initialisant t peuvent être déplacées en dehors de R_j, il faut insérer dans R_j des instructions qui décrémentent t à la sortie de R_i. Ceci afin que chaque fois que l'on réexécutera R_i à partir de R_j, t possède la valeur requise avant l'entrée dans R_i.

Si l'on peut réduire des instructions dans R_j qui dépendent d'une variable récursivement assignée dans R_j, il peut être intéressant, afin de limiter la place mémoire, de réutiliser t, mais pour ce faire il faut soustraire de t la valeur qui correspond à une incrémentation répétitive de t dans R_i. Dès lors, il est nécessaire de connaître l'évolution totale de t dans R_i.

Exemple



Légende



: segment fictif inséré

On ne pouvait pas insérer le code de décrémentation de t dans le segment ④ car celui-ci a plus d'un prédécesseur; ceci serait possible si ④ n'avait que ③ comme prédécesseur immédiat (sortie de la région R).

5.9.3. Conditions pratiques pour qu'une variable t soit réinitialisable

Pour qu'une variable t générée lors de la réduction des instructions d'une région R soit réinitialisable, nous imposerons les conditions suivantes sur la région R :

1) R ne peut avoir qu'un seul prédécesseur

2) R ne peut avoir qu'un seul segment de sortie.

Il est cependant permis que R possède plusieurs segments d'entrée et plusieurs segments successeurs.

Les raisons de ces restrictions seront plus compréhensibles lorsque nous aurons énoncé les restrictions relatives à la variable I récursivement assignée de laquelle dépend t. Ces restrictions sont les suivantes :

11) I doit être récursivement assigné dans le segment de sortie de R, uniquement;

22) le test se trouvant dans le segment de sortie et qui, s'il est vérifié ou non vérifié, nous permet de sortir de la région, doit porter sur la valeur de I;

33) la variable I doit être initialisée dans le segment prédécesseur de la région par une valeur entière a (a = constante);

44) l'assignation récursive doit être de la forme I := I + c où c est un entier positif ou négatif (c = constante);

55) dans le test, la valeur de I doit être comparée à une valeur entière b (b = constante).

Les raisons de ces restrictions sont maintenant examinées.

1°) <u>1) et 11) combinés</u>

On veut s'assurer que la valeur de I avant de rentrer dans la région R est toujours la même, et ceci d'une façon simple, c'est-à-dire ne demandant pas trop d'analyse à la compilation.

2°) <u>2) et 22) combinés</u>

Lorsqu'il n'y a qu'un seul segment de sortie dans une région, ce

segment est sûrement un point d'articulation et par conséquent l'assignation récursive est certainement exécutée. De plus, on est sûr d'incrémenter t d'une valeur entière un certain nombre de fois.

3°) 33), 44) et 55) se justifient de la façon suivante :

- L'exigence que a, b et c soient des constantes, a pour objectif le calcul immédiat à la compilation de la valeur de n (le nombre de fois que t a été incrémenté par l'exécution répétitive de la région).

- On peut sans doute relâcher ces restrictions en exigeant simplement que a, b et c soient des constantes de régions entières.

- Le type entier doit toujours être maintenu comme restriction, ceci permettant un calcul exact de n.

Soient a, b et c les constantes entières signalées plus haut. Nous pouvons maintenant calculer n. Pour rappel, nous donnons ici la forme des instructions initialisant I et incrémentant I, à savoir : I := a, I := I + c.

1° Si $a \leq b$, si c est positif et si le test de sortie de R donne le contrôle à R, si $I \leq b$, alors $n = \text{entier}\left(\frac{b-a}{c}\right) + 1$.

2° Si $a \leq b$, si c est positif et si le test de sortie de R donne le contrôle à R si $I < b$, alors

1 $n = \frac{b-a}{c} + 1$ si le reste de la division entière est non nul;

2 $n = \frac{b-a}{c}$ si le reste de la division entière est nul.

3° Si $a \geqslant b$, si c est négatif et si le test de sortie de R donne le contrôle à R si $I \geqslant b$, alors $n = \frac{b-a}{c} + 1$.

Il existe plusieurs autres cas pour lesquels n peut être calculé d'une façon identique.

Par exemple, si $a \geqslant b$, si c est positif et si le test de sortie de R donne le contrôle à R si $I \geqslant b$, alors on peut affirmer que le programme bouclera indéfiniment s'il passe et entre dans R.

Ceci pourrait être signalé au programmeur par un message d'erreur.

Une fois la valeur n déterminée et donc les conditions sur I ont été vérifiées, il est possible de déterminer aisément si une variable générée t est réinitialisable. Il suffit de tester si l'incrément D de t est entier.

En effet, si cet incrément n'était pas entier, on ne pourrait pas réinitialiser t à la valeur requise. La réinitialisation de t est en quelque sorte une décrémentation de t, t := t - n * D.

Cette instruction t := t - n * D est insérée juste après le segment de sortie de la région R et juste avant tous les successeurs de R, si l'une des conditions suivantes est vérifiée :

1° Les instructions initialisant t à la valeur requise pour entrer dans une région R_i couverte par la région R_j ont été déplacées en dehors de R_j.

2° Dans la région R_j couvrant R_i, t a été réutilisé pour la réduction de certaines instructions de R_j.

5.9.4. Implication de la réinitialisation sur la réduction des opérateurs

Cette implication concerne le choix de la variable temporaire à générer lorsqu'une instruction termine une séquence d'instructions réductibles.

Si l'instruction qui termine une séquence d'instructions réductibles est une assignation à une variable temporaire t réinitialisable, cette variable t est utilisée pour exprimer la réduction.

5.10. Remarques

Nous résumerons ici quelques points importants avant d'aborder l'étude des techniques d'optimisation qu'il nous reste à envisager.

1) Le segment fictif contient maintenant toutes les instructions qui ont été déplacées en arrière et toutes les instructions initialisant des variables temporaires qui ont été générées lors de la réduction des opérateurs.

2) Tous les incréments non directement calculables des variables temporaires générées sont calculés par une séquence d'instructions qui ont été placées dans le segment fictif.

3) Les définitions récursives (assignations) des variables temporaires générées ont toutes été placées après les assignations récursives qui ont donné lieu aux réductions.

4) Si une région R_i couvre une région R_j et si R_j a été traitée par l'optimiseur, celui-ci dispose d'une table qui lui donne des informations sur les variables réinitialisables de R_j pour lesquelles les instructions de décrémentations doivent être générées après la sortie de R_j.

5) Lorsque R_i sera examiné, une table de variables réinitialisables sera constituée. Cette table aura une entrée par variable réinitialisable, entrée qui mentionnera l'identificateur de la variable et la valeur du décrément.

6) Chaque instruction qui termine une séquence d'instructions réductibles dépend maintenant de la variable temporaire qui exprime la réduction de la séquence.

Chapitre VI

TRANSFORMATION DE TESTS ET ELIMINATION D'ASSIGNATIONS "MORTES"

Dans ce chapitre, nous envisageons deux techniques d'optimisation intéressantes, mais qui ne sont pas vitales dans un compilateur optimiseur.

La transformation de tests dépend très fortement de la forme intermédiaire que l'on a choisie.

L'élimination d'assignations "mortes", qui est sans doute la plus intéressante des deux, permet en quelque sorte de corriger les erreurs d'inattention d'un programmeur.

6.1. Transformation de tests

6.1.1. Remarques préliminaires

Certains auteurs * proposent une technique d'optimisation transformant les tests de sortie d'une région R.

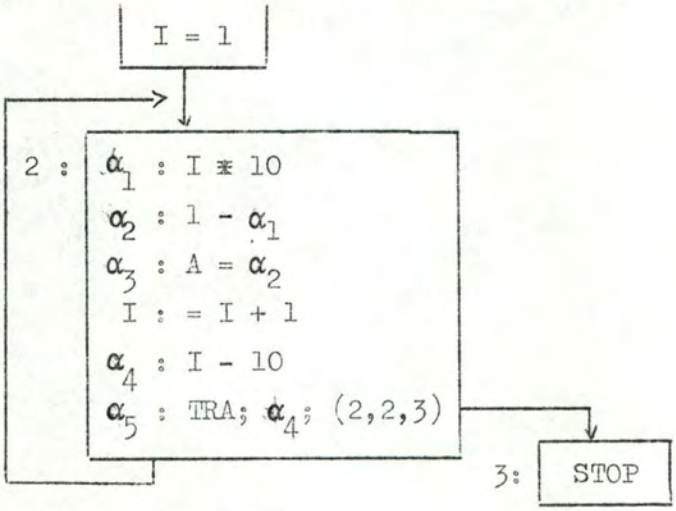
Des tests faisant intervenir des variables-programmeur récursivement assignées peuvent parfois être remplacés par des tests portant sur les variables générées lors de la réduction des opérateurs.

---

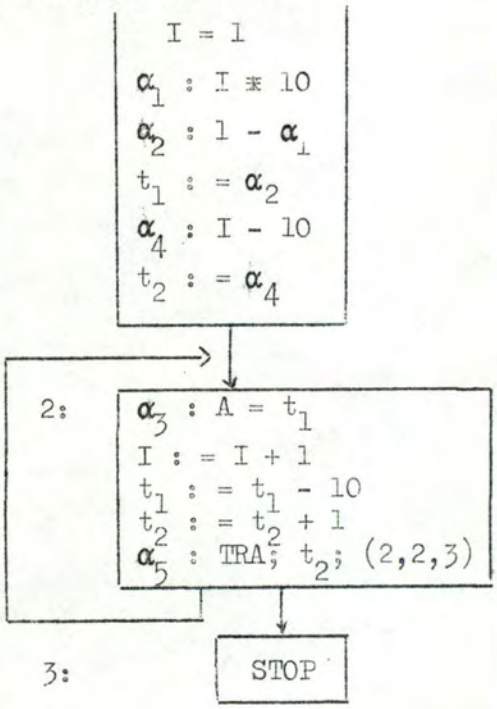
* Allen [ALLEN 69], Lowry et Medlock [L M 69], Cocke [COCKE 70].

Généralement, les variables-programmeur récursivement assignées ne sont utilisées que dans des expressions réductibles et dans les tests de sortie d'une région. En réduisant ces expressions et en remplaçant ces tests par des tests portant sur la ou les variables exprimant les réductions, il est parfois possible d'éliminer du programme les variables-programmeur récursivement assignées.

Mais si la forme intermédiaire est suffisamment raffinée, nous pouvons nous passer de cette technique. Nous le montrerons sur l'exemple suivant: le texte intermédiaire sera transformé par la réduction des opérateurs



pour donner le résultat suivant :



La transformation de test a été automatiquement réalisée par la réduction des opérateurs grâce à l'introduction d'une nouvelle variable t_2. Une forme intermédiaire aussi raffinée que la nôtre pose parfois des difficultés lorsque l'on veut profiter de la notion de variable réinitialisable.

Dans le deuxième paragraphe de cette section, nous exposerons une méthode permettant de réaliser la transformation de tests pour une forme intermédiaire moins raffinée.

Remarquons ici que si l'on garde le texte intermédiaire proposé dans le premier chapitre, il est intéressant que les routines sémantiques traduisent sous une forme particulière les expressions-sources faisant intervenir des opérateurs de comparaison arithmétique.

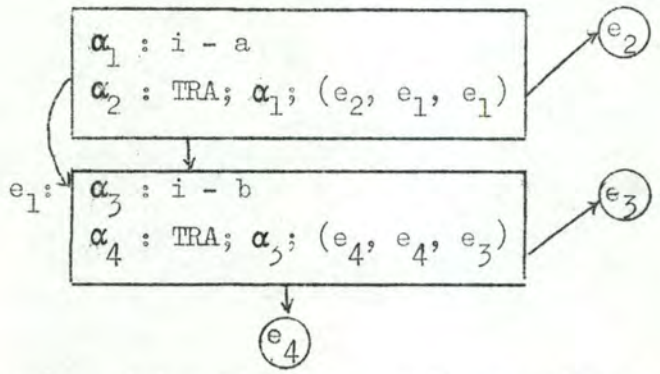
Exemples

1) <u>if</u> $i \leq a$ <u>then</u> <u>goto</u> e_2 <u>else</u> <u>goto</u> e_3 devrait être traduit en

α_1 : i - a

α_2 : TRA; α_1; (e_2, e_2, e_3)

2) <u>if</u> $i < a$ or $i > a$ <u>then</u> <u>goto</u> e_2 <u>else</u> <u>goto</u> e_3 devrait être traduit en



Ceci permet la réduction de l'instruction i - a et de l'instruction i - b.

L'exemple (2) dégage aussi une technique de simplification des expressions booléennes. Par la traduction de <u>if</u> $i < a$ or $i > b$ <u>then</u> <u>goto</u> e_2 <u>else</u> <u>goto</u> e_3 en deux tests successifs, il est possible de diminuer en moyenne le temps d'exécution du programme.

Des améliorations sensibles peuvent être apportées à un programme par le programmeur qui appliquerait le théorème de De Morgan pour la simplification des expressions logiques.

Il pourrait également profiter de sa connaissance du problème programmé pour arranger de façon optimale la cascade de tests, mais sans doute en rendant les programmes moins lisibles.

6.1.2. Remplacement de tests

6.1.2.1. L'idée est de remplacer un test du type IF (I - X) par un test IF (t - X') où t est la variable temporaire générée lors de la réduction d'instruction utilisant la variable récursivement assignée I. Pour ce faire, il faut résoudre plusieurs problèmes.

1° Pour une variable récursivement assignée I, plusieurs variables temporaires t peuvent être générées lors de la réduction des opérateurs. Un choix doit donc être fait parmi ces variables t.

2° Si X est la valeur à laquelle I est comparée, cette valeur doit être remplacée par une valeur X' qui est obtenue en remplaçant I par X dans la séquence initialisant t. Cette séquence est linéaire; ceci découle de la réduction des opérateurs.

Exemple

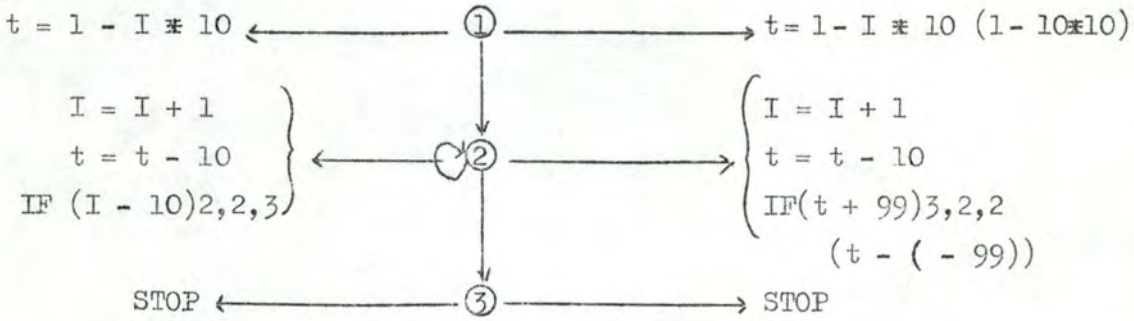
I = 1	1	I = 1 t = I * 10	I = 1 t = I * 10
α_1 I * 10 A = α_1 I = I + 1 IF (I - 10)2,2,3	2	A = t I = I + 1 t = t + 10 IF (I - 10)2,2,3	A = t I = I + 1 t = t + 10 IF (t - 100)2,2,3
STOP	3	STOP	STOP

3° Lorsque les incréments de I et de t diffèrent en signe, l'instruction de branchement doit être légèrement modifiée.

Exemple

si I = I + D_1 $D_1 > 0$ et si t = t - D_2 $D_2 > 0$ alors l'instruction IF (I - X) a, b, c doit être changée en IF (t - X') c, b, a.

Exemple



6.1.2.2. Procédure permettant de réaliser la transformation de tests

Après avoir réduit toutes les séquences d'instructions réductibles de la région R qui utilisait la variable I récursivement assignée dans R :

1° Tous les segments de sortie de R sont cherchés.

2° Tous les tests portant sur la valeur de I sont localisés.

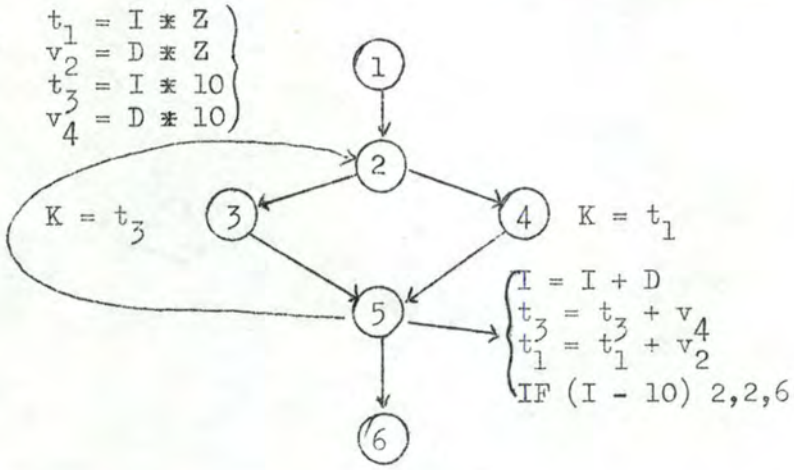
3° Toutes les variables temporaires t générées pour la réduction des séquences d'instructions réductibles utilisant I sont listées. Soit (t_1, ..., t_n) la liste de ces variables.

4° Soit $f_1(I)$, ..., $f_n(I)$ la liste des séquences telle que $t_1 = f_1(I)$, ..., $t_n = f_n(I)$ soient les initialisations des variables temporaires. Les f_i sont linéaires.

5° Soit IF (I - X) un test donné.

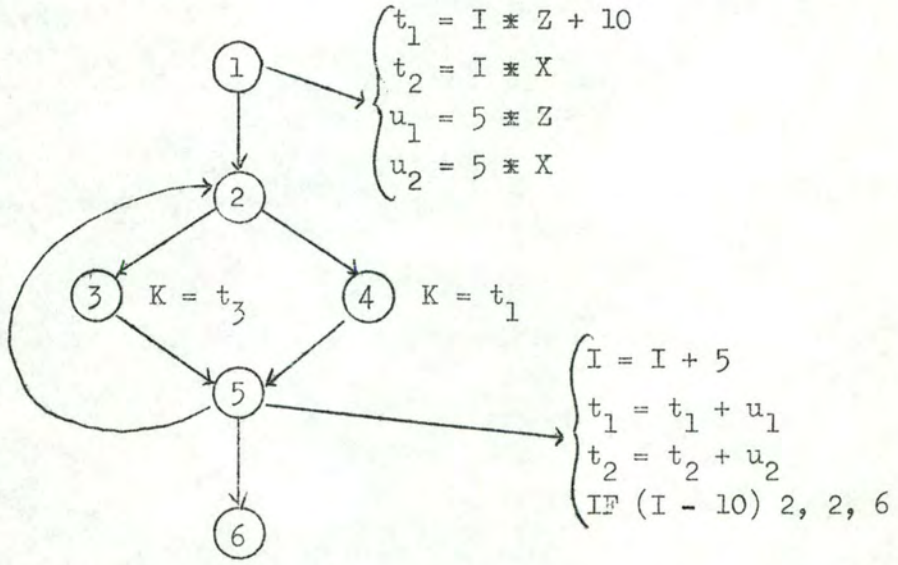
5° (1) Si X est une constante, le t_i choisi pour remplacer I dans le test est celui pour lequel $f_i(X) = X'$ est une constante. Le test devient IF (t_i - X') et l'instruction de branchement est modifiée si nécessaire.

Exemple

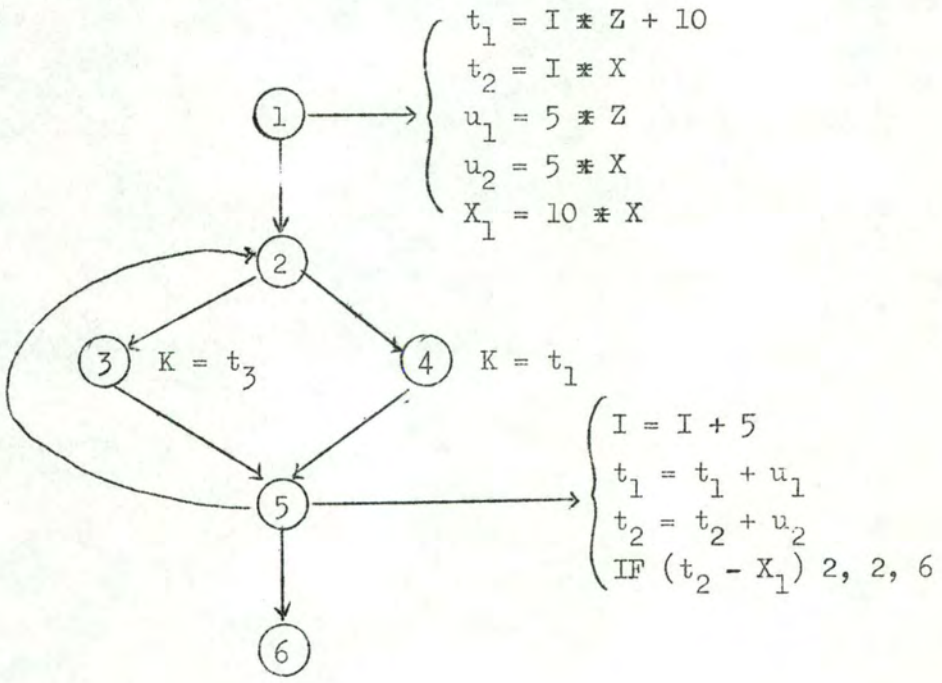


- $f_1(I) = I * Z$ et $f_2(I) = I * 10$; dès lors $f_1(10)$ n'est pas une constante tandis que $f_2(10) = 100$; nous choisirons donc t_3 pour remplacer I dans le test IF (I - 10) 2, 2, 6. Ce dernier deviendra IF (t_3- 100) 2, 2, 6.

<u>Exemple</u>



$f_1(I) = I * Z + 10$, $f_2(I) = I * X$. Aucune des séquences $f_1(10)$, $f_2(10)$ ne délivre une valeur connue statiquement. Or $f_2(10)$ a un temps minimal de calcul d'où t_2 sera choisi pour remplacer I.



Si, quel que soit i, $f_i(X)$ n'est pas une constante, alors le t_j choisi est celui correspondant au $f_j(X)$ ayant un temps machine de calcul minimal. La séquence $X' = f_j(X)$ est insérée dans le segment fictif et le test devient IF (t_j - X'); l'instruction de branchement est modifiée si nécessaire.

Nous pouvons remarquer que, lorsque $f_j(X)$ est traitée par la propagation de constante - ce qui a lieu lorsque le segment fictif est traité par la propagation de constante - la séquence $f_j(X)$ est fortement réduite.

5° (2) Si X n'est pas une constante mais est néanmoins une constante de région, alors le t_j choisi est celui correspondant au $f_j(X)$ ayant un temps machine de calcul minimal. La séquence $X' = f_j(X)$ est insérée dans le segment fictif et le test devient IF (t_j - X'). L'instruction de branchement est modifiée si nécessaire.

(Voir exemple page précédente)

Remarques

1. Nous écrivons le temps machine de calcul minimal car il n'est pas nécessairement vrai que le $f_j(X)$ comportant le moins d'opérateurs soit le $f_j(X)$ minimal du point de vue du temps de calcul.

2. Le remplacement de tests est une optimisation qui expose un plus grand nombre de variables candidates à une élimination. En réalité, cette optimisation n'est pas essentielle.

6.2. Elimination d'assignations "mortes"

6.2.1. Définition

Une assignation est dite "morte" si le résultat de l'assignation n'est jamais utilisé dans le programme ou si l'assignation est récursive et constitue la seule utilisation de son résultat.

Une assignation "morte" peut être éliminée du programme. De même que les instructions n'étant utilisées par aucune autre instruction et figurant dans la séquence d'évaluation de l'expression arithmétique dont le résultat est l'"objet" de l'assignation "morte".

Exemple

α_1 : A * B
α_2 : α_1 + C
α_3 : C * D
α_4 : $\alpha_2 + \alpha_3$
α_5 : A : = α_4
α_6 : A * α_1
α_7 : Z : = α_6

Supposons que α_5 soit une assignation "morte". Dès lors, α_5 peut être éliminé ainsi que α_2, α_3, α_4, à condition toutefois que α_2, α_3 et α_4 ne soient pas utilisés autre part dans le segment. Par contre, α_1 ne peut pas être éliminé.

Les assignations "mortes" ont plusieurs causes, à savoir :

1° Une erreur de programmation ou une programmation peu soignée. Ceci dépend parfois de la structure du langage. Ainsi le langage FORTRAN 4 n'oblige pas le programmeur à déclarer explicitement les quantités utilisées dans le programme et favorise, nous semble-t-il, ce genre d'erreur. D'autres langages, tels que ALGOL 60 et COBOL, ont l'avantage d'obliger le programmeur à déclarer les quantités utilisées. Ceci, nous semble-t-il, force le programmeur à une prise de conscience de la place mémoire qu'il utilise.

2° La réduction des opérateurs, c'est-à-dire la réduction de séquences d'instructions réductibles, et le remplacement de tests par des tests portant sur les variables générées durant cette phase de réduction exposent des assignations récursives à être déclarées "mortes".

6.2.2. Application de la définition

Pour une certaine assignation d appartenant à un segment S donné, il faut analyser toutes les instructions suivant l'assignation dans le segment et toutes les instructions se trouvant dans les segments successeurs de S pour déterminer si d est morte ou pas.

Nous commençons par examiner les instructions suivant d dans S. Si parmi elles se trouve une utilisation du résultat de d, alors d n'est pas "mort" et nos recherches peuvent être stoppées.

Dans le cas contraire, deux possibilités se présentent :

1) S est un successeur immédiat de lui-même: s'il existe une utilisation de la variable v assignée par d, dans les instructions précédant d dans S, nous pouvons dire que d n'est pas "morte".

2) S n'est pas un successeur immédiat de lui-même: il convient alors d'examiner les segments situés sur les chemins issus de S.

Si, sur un chemin issu de S, nous trouvons une nouvelle assignation à la variable v sans avoir précédemment rencontré d'utilisation de v, ou si, dans les mêmes conditions, nous atteignons un segment de sortie du programme, nous pouvons arrêter nos recherches sur ce chemin pour envisager un autre chemin issu de S.

Si, après l'examen de tous les chemins issus de S, nous n'avons jamais trouvé d'utilisation de v avant une assignation à v ou avant un segment de sortie, nous pouvons conclure que d est "morte".

Si, sur un chemin quittant S, nous trouvons une utilisation de v avant une assignation à v ou avant une sortie du programme, nous pouvons conclure immédiatement que d n'est pas "morte".

Une fois d déclarée "morte", nous pouvons éliminer l'assignation, ainsi que la séquence d'instructions (ou certaines instructions dans cette séquence) qui produit la valeur que l'on assigne.
Il faut bien entendu que les instructions ne soient pas utilisées par d'autres instructions dans S.

Remarques

- La détection d'"assignations mortes" et l'élimination de ces dernières est la dernière optimisation effectuée par l'optimiseur. Elle est aussi la plus globale.

- Il est nécessaire de développer une procédure qui ne prenne pas trop de temps. On aura intérêt à réduire des opérations de cheminement dans un graphe à des opérations booléennes.

6.2.3. Procédure d'élimination d'assignations "mortes"

Nous envisagerons deux cas, à savoir :

1° les assignations récursives;

2° les autres assignations.

Nous supposons que nous disposons :

1° de la matrice booléenne C associée au graphe G du programme;

2° des tables ASS, REF et RAA;

3° d'une table VRA (Variable Récursivement Assignée).

Pour les définitions des tables ASS, REF, RAA, nous renvoyons le lecteur au chapitre III.

VRA est la table des variables (programmeur et temporaires) récursivement assignées et des segments d'instructions contenant les assignations récursives, mais dans lesquels la seule référence à la variable doit être l'assignation récursive.

Rappelons que les tables RAA, REF et ASS ont été mises à jour lors des différentes phases de l'optimisation.

6.2.3.1. Les assignations récursives

Soient v_1, v_2, ..., v_m les différentes entrées dans VRA.
Pour des raisons de facilité, VRA est représenté sous une forme booléenne:
VRA $[v_i, j] = 1$ si v_i satisfait aux conditions énoncées dans le chapitre V et si, de plus, la seule référence à v_i dans le segment j est l'assignation récursive.

Introduisons trois vecteurs booléens de travail **S**, X et Y, dont la dimension, soit n, est égale au nombre de segments dans le programme.

Soient v_i une variable et j un segment d'instructions tels que VRA $[v_i, j] = 1$. S $[k] = 1$ si le segment k est un successeur de j, non encore examiné par la procédure d'élimination des assignations récursives "mortes"; S $[k] = 0$ si k n'est pas un successeur de j ou si k a déjà été examiné.
X $[k] = 1$ si le segment k est un successeur de j déjà traité par la procédure d'élimination des assignations récursives "mortes"; sinon X $[k] = 0$.
Y $[k] = 1$ si ASS $[v_i, k] = 0$ et S $[k] = 1$, c'est-à-dire si le segment k, non

encore examiné par la procédure, est un successeur du segment j ne contenant pas d'instruction modifiant la valeur de la variable v_i. Y [k] = 0 dans le cas contraire.

(1) i := 1.

(2) Pour v_i, sélectionner la ligne VRA $[v_i, \emptyset]$.

(3) j := 1.

(4) <u>Si</u> VRA $[v_i, j]$ = 1 <u>alors</u> aller en (5)
<u>sinon</u> aller en (11)

(5) Le vecteur booléen S est initialisé avec les successeurs de j.
S := C $[j, \emptyset]$.

(6) X est initialisé de la manière suivante : X [j] : = 1, X [k]:= 0 pour tout $k \in \{1, \ldots, n\} \setminus \{j\}$.
Vu la construction de VRA, nous pouvons affirmer que j ne contient aucune référence à v_i autre que l'assignation récursive elle-même.

<u>Notations</u> : 1 = true, 0 = false.
Un vecteur booléen non nul est un vecteur ayant au moins une composante égale à 1.

(7) z := S $\wedge$ RAA $[v_i, \emptyset]$ est évalué.
<u>Si</u> z est un vecteur non nul, <u>alors</u> il existe un segment successeur de j qui contient une référence* à v_i, avant une instruction modifiant la valeur de v_i. Dès lors, l'assignation récursive figurant dans le segment j n'est pas "morte". Dans ces conditions, nous allons en (11).
<u>Si</u> z est un vecteur nul, <u>alors</u> nous allons en (8).

(8) z étant un vecteur nul, nous pouvons affirmer que dans tous les segments successeurs de j, non encore examinés, il ne figure pas de référence à v_i avant une instruction modifiant la valeur de v_i. Dès lors, il faut envisager les successeurs des segments figurant dans la liste S. Parmi ceux-ci, certains segments, ou peut-être tous, sont susceptibles de contenir une instruction modifiant la valeur de v_i. Pour déterminer les segments dans lesquels <u>ne figure</u> pas d'instruction modifiant la valeur de v_i, Y := S $\wedge \overline{\text{ASS}}$ $[v_i, \emptyset]$ est évalué.

---

* "référence à v_i" est synonyme de "utilisation de v_i".

<u>Si</u> Y est un vecteur nul, <u>alors</u> tous les successeurs du segment j, figurant dans la liste S, contiennent au moins une instruction modifiant la valeur de v_i et ceci avant une instruction référençant v_i. Dans ces conditions, l'assignation récursive figurant dans le segment j est "morte" et peut être éliminée. Nous allons alors en (11).

<u>Si</u> Y est un vecteur non nul, <u>alors</u> nous allons en (9).

(9) Nous envisageons maintenant les successeurs, non encore examinés, des segments figurant dans la liste Y, c'est-à-dire les segments k pour lesquels $Y[k] = 1$.

<u>Si</u> aucun d'eux ne possède de successeurs, <u>alors</u> Y est une liste de points de sortie du programme et l'assignation récursive figurant dans le segment j est "morte" et peut être éliminée. Nous allons en (11). Si k est un segment de sortie, nous forçons $Y[k]$ à zéro.

<u>Si</u> les segments figurant dans la liste Y possèdent des successeurs, <u>alors</u> le vecteur X est mis à jour par $X := X \vee S$ et un nouveau vecteur S est généré de la façon suivante :

1) $S[k] := 0$ pour $k = 1, \ldots, n$,
2) pour $k = 1, \ldots, n$, <u>si</u> $Y[k] = 1$ <u>alors</u> $S := S \vee C[k,\emptyset]$ est calculé, et nous allons en (10).

(10) S représente maintenant la liste de tous les successeurs immédiats des segments figurant dans l'ancienne liste S et pour laquelle $Y[k] = 1$. Parmi ces segments, certains ont déjà peut-être été examinés. Afin de les éliminer, $S := S \wedge \overline{X}$ est calculé.

<u>Si</u> S est un vecteur non nul, <u>alors</u> nous allons en (7).

<u>Si</u> S est un vecteur nul, <u>alors</u> tous les segments successeurs du segment j ont été examinés et, dans aucun de ces segments, il n'a été trouvé d'instruction référençant v_i avant une instruction modifiant la valeur de v_i. Dès lors, l'assignation récursive figurant dans le segment j est "morte" et peut être éliminée. Nous allons en (11).

(11) $j := j + 1$.

<u>Si</u> $j > n$ <u>alors</u> nous allons en (12)
<u>sinon</u> nous allons en (4).

(12) i := i + 1.

<u>Si</u> i > m <u>alors</u> nous terminons l'algorithme
<u>sinon</u> nous allons en (2).

6.2.3.2. <u>Les autres assignations</u>

Pour ces assignations, nous ne possédons plus de table VRA indiquant le segment dans lequel se trouve l'assignation récursive candidate à l'élimination. Nous perdons du fait même un renseignement précieux, à savoir que la seule référence à la variable récursivement assignée, dans le segment où figure l'assignation récursive, est l'assignation elle-même.
Nous rencontrons donc une difficulté supplémentaire: dans un même segment peuvent figurer plusieurs assignations à une même variable. Nous sommes dès lors obligés de travailler d'abord au niveau restreint des segments d'instructions et ensuite au niveau global du programme.

6.2.3.2.1. Au niveau du segment d'instructions, plusieurs remarques peuvent être faites.

A la simple lecture d'un segment d'instructions, on peut déjà dire qu'une assignation d à une variable v n'est pas "morte" si, après d, il figure, dans le segment, une instruction r référençant la variable v et si entre d et r il ne figure pas d'autres instructions modifiant la valeur de v.
De plus, une assignation d à une variable v est "morte" si, après d, il figure, dans le segment, une autre assignation d' à v et si, entre d et d', il n'existe pas d'instructions référençant v.

Une assignation d à une variable v est candidate à une élimination d'assignation "morte" si et seulement si, après d, il ne figure dans le segment aucune autre assignation d' à v et aucune instruction référençant la variable v.

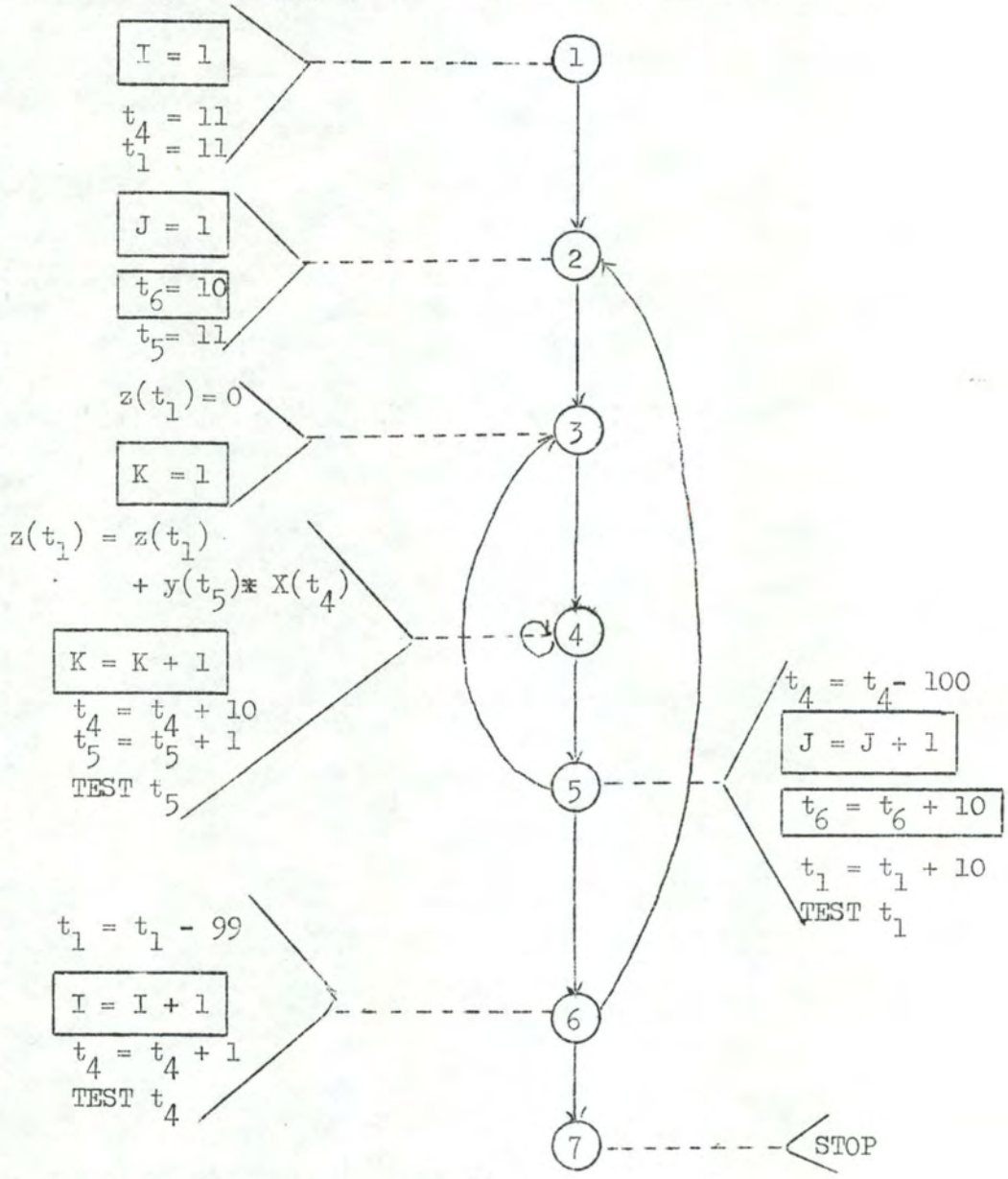
Les seules assignations qui sont candidates à une élimination d'assignations "mortes" seront traitées par une procédure similaire à la procédure décrite en 6.2.3.1.

6.2.3.2.2. Des remarques ci-dessus nous déduirons la procédure suivante.

Soit n le nombre de segments formant le programme P. Pour une assigna-

Exemple de table VRA

Soit le programme suivant :



VRA

I	6
J	5
K	4
t_4	5
t_6	5
t_1	6

Légende : Les assignations entourées sont des assignations mortes.

Pour ce programme, nous avons la matrice C et les tables ASS, REF, RAA suivantes :

ASS	1	2	3	4	5	6	7
I	1	0	0	0	0	1	0
J	0	1	0	0	1	0	0
K	0	0	1	1	0	0	0
t_1	1	0	0	0	1	1	0
t_4	1	0	0	1	1	1	0
t_5	0	1	0	1	0	0	0
t_6	0	1	0	0	1	0	0

REF	1	2	3	4	5	6	7
I	0	0	0	0	0	1	0
J	0	0	0	0	1	0	0
K	0	0	0	1	0	0	0
t_1	0	0	1	1	1	1	0
t_4	0	0	0	1	1	1	0
t_5	0	0	0	1	0	0	0
t_6	0	0	0	0	1	0	0

RAA	1	2	3	4	5	6	7
I	0	0	0	0	0	1	0
J	0	0	0	0	1	0	0
K	0	0	0	1	0	0	0
t_1	0	0	1	1	1	1	0
t_4	0	0	0	1	1	1	0
t_5	0	0	0	1	0	0	0
t_6	0	0	0	0	1	0	0

VRA	1	2	3	4	5	6	7
I	0	0	0	0	0	1	0
J	0	0	0	0	1	0	0
K	0	0	0	1	0	0	0
t_1	0	0	0	0	0	1	0
t_4	0	0	0	0	1	0	0
t_5	0	0	0	0	0	0	0
t_6	0	0	0	0	1	0	0

C	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0
3	0	0	0	1	0	0	0
4	0	0	0	1	1	0	0
5	0	0	1	0	0	1	0
6	0	1	0	0	0	0	1
7	0	0	0	0	0	0	0

<u>Exemple d'élimination d'une assignation récursive "morte"</u>, pour I = I + 1 dans le segment 6.

$S = [0\ 1\ 0\ 0\ 0\ 0\ 1]$

$X = [0\ 0\ 0\ 0\ 0\ 1\ 0]$

$S \wedge RAA\ [I,\emptyset] = [0\ 1\ 0\ 0\ 0\ 0\ 1] \wedge [0\ 0\ 0\ 0\ 0\ 1\ 0] = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$Y = S \wedge \overline{ASS}\ [I,\emptyset] = [0\ 1\ 0\ 0\ 0\ 0\ 1] \wedge [0\ 1\ 1\ 1\ 1\ 0\ 1] = [0\ 1\ 0\ 0\ 0\ 0\ 1]$

$X = X \vee S = [0\ 0\ 0\ 0\ 0\ 1\ 0] \vee [0\ 1\ 0\ 0\ 0\ 0\ 1] = [0\ 1\ 0\ 0\ 0\ 1\ 1]$

$S = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$S = S \vee (C_{2\emptyset} \vee C_{7\emptyset}) = [0\ 0\ 1\ 0\ 0\ 0\ 0]$

$S = S \wedge \overline{X} = [0\ 0\ 1\ 0\ 0\ 0\ 0] \wedge [1\ 0\ 1\ 1\ 1\ 0\ 0] = [0\ 0\ 1\ 0\ 0\ 0\ 0]$

$S \wedge RAA\ [I,\emptyset] = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$Y = S \wedge \overline{ASS}\ [I,\emptyset] = [0\ 0\ 1\ 0\ 0\ 0\ 0]$

$X = X \vee S = [0\ 1\ 1\ 0\ 0\ 1\ 1]$

$S = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$S = S \vee C_{3\emptyset} = [0\ 0\ 0\ 1\ 0\ 0\ 0]$

$S = S \wedge \overline{X} = [0\ 0\ 0\ 1\ 0\ 0\ 0]$

$S \wedge RAA\ [I,\emptyset] = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$Y = [0\ 0\ 0\ 1\ 0\ 0\ 0]$

$X = [0\ 1\ 1\ 1\ 0\ 1\ 1]$

⋮

$S = [0\ 0\ 0\ 1\ 1\ 0\ 0]$

$S = [0\ 0\ 0\ 0\ 1\ 0\ 0]$

$S \wedge RAA\ [I,\emptyset] = [0\ 0\ 0\ 0\ 0\ 0\ 0]$

$Y = [0\ 0\ 0\ 0\ 1\ 0\ 0]$

$X = [0\ 1\ 1\ 1\ 1\ 1\ 1]$

⋮

$S = [0\ 0\ 1\ 0\ 0\ 1\ 0]$

$S = [0\ 0\ 0\ 0\ 0\ 0\ 0] \longrightarrow$ fin I = I + 1 est uns assignation récursive "morte" et peut donc être éliminée.

6.2.3.3. Remarques

1° Chaque fois qu'une assignation est éliminée, il faut modifier en conséquence les tables RAA, REF, ASS.

2° On peut traiter le cas de plusieurs assignations récursives à la même variable v dans un même segment en utilisant la procédure 6.2.3.2.2.

3° On commence par éliminer les assignations récursives "mortes" et puis on élimine les autres. Ceci n'est pas impératif.

4° Remarquons la différence entre instruction modifiant la valeur d'une variable et assignation. Nous n'avons envisagé que l'élimination des assignations et non pas des instructions modifiant la valeur d'une variable v. Ce dernier cas est beaucoup plus complexe (exemple: l'élimination d'appel de procédure).

5° L'élimination d'assignations "mortes" est sans doute l'une des seules techniques d'optimisation qui permette à la fois un gain de place mémoire et un gain de temps.

6° Il est intéressant de munir l'optimiseur d'un système de gestion de la table des variables simples (V.S.) similaire au système de gestion de la table des constantes (T.C.).

7° Nous n'avons pas pris en considération l'élimination d'assignations ayant pour cibles des variables indicées. Ce problème compliquerait les procédures d'élimination car nous ne disposons pas de tables RAA, REF, ASS pour des variables indicées.

Chapitre VII

INSERTION DU SEGMENT FICTIF DANS LE PROGRAMME

Les instructions déplacées en arrière d'une région ou générées lors de la réduction des opérateurs ou du remplacement de tests doivent être insérées dans le programme. Pendant l'optimisation d'une région, ces instructions ont été déposées dans un segment fictif. Après l'optimisation de cette région, elles doivent être insérées dans le programme de telle sorte que, quel que soit le chemin suivi, elles soient exécutées avant l'entrée dans la région. Ce chapitre décrit succinctement l'insertion du segment fictif dans le programme.

7.1. Critères d'insertion

Le problème qui se pose est de garantir l'exécution des instructions se trouvant dans le segment fictif avant d'entrer dans la région pour laquelle ces instructions ont été déplacées ou générées. A cet effet, nous pouvons soit insérer les instructions composant le segment fictif, dans les segments prédécesseurs immédiats de la région, soit former un véritable segment de ces instructions et l'insérer, comme prédécesseur immédiat de la région, sur tous les chemins entrant dans la région. La façon dont ces instructions sont insérées dans le programme s'appuie sur les critères suivants.

1) Si le segment S prédécesseur immédiat de la région R ne possède qu'un seul successeur immédiat (a fortiori un point d'entrée de R), les instructions formant le segment fictif sont insérées dans le segment S juste avant l'instruction de branchement.

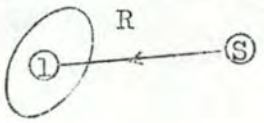
2) Si le segment S prédécesseur immédiat de la région R possède plusieurs successeurs immédiats qui sont tous des points d'entrée de R, alors les instructions formant le segment fictif sont insérées dans le segment S juste avant l'instruction de branchement. Il faut que les instructions figurant dans le segment fictif n'affectent pas le branchement.

3) Si le segment S prédécesseur immédiat de la région R possède plusieurs successeurs immédiats qui ne sont pas tous des points d'entrée de R ou si son instruction de branchement peut être affectée par les instructions figurant dans le segment fictif, alors les instructions formant le segment fictif sont érigées en véritable segment (segment érigé) qui, suivant les cas, est inséré ou pas dans le programme :

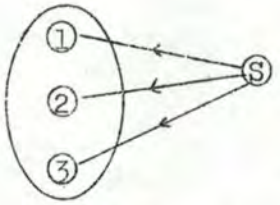
1° Si le point d'entrée vers lequelbranche S est déjà le successeur immédiat d'un segment érigé, alors l'instruction de branchement de S est modifiée pour brancher vers ce segment érigé.

2° Si le point d'entrée vers lequel branche S n'est pas encore le successeur immédiat d'un segment érigé, alors le segment érigé est inséré sur l'arc allant de S vers le point d'entrée de R. L'instruction de branchement de S est modifiée en conséquence.

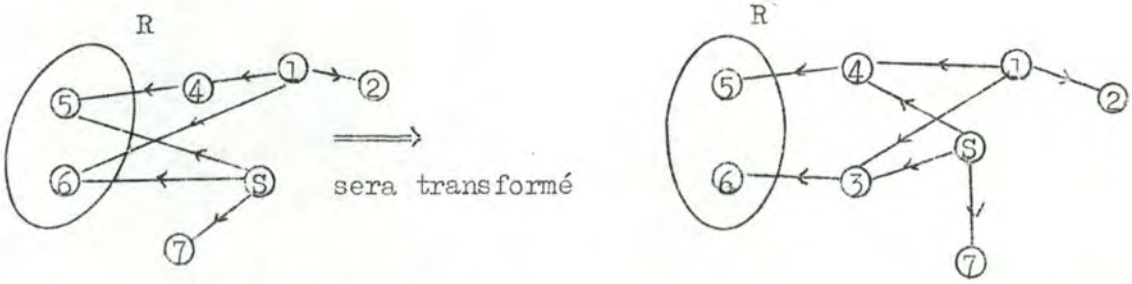
Illustration des trois critères ci-dessus



Le segment fictif est inséré dans le segment S



Le segment fictif est inséré dans le segment S



④ est un segment érigé pour R. ③ est le nouveau segment érigé

Les critères 1, 2 et 3 permettent l'insertion des instructions formant le segment fictif sur base d'une petite quantité d'informations locales. En effet, nous ne faisons appel qu'aux notions de prédécesseur immédiat d'une région R et de successeur immédiat d'un segment.

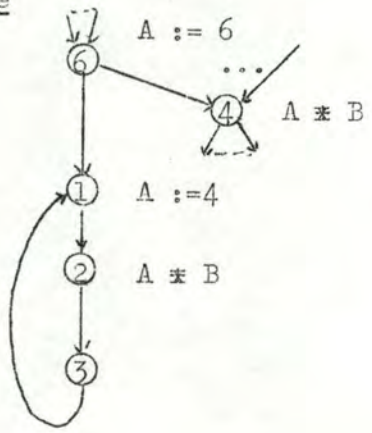
Sans faire appel à la notion de lien LAR (lien assignation référence), les critères ci-dessus garantissent une non rupture de ces liens. Ceci est dû au fait que nous avons restreint les segments prédécesseurs dans lesquels les instructions étaient insérées à ceux dont tous les successeurs immédiats sont des points d'entrée.

De plus, les instructions insérées ne seront pas exécutées si l'on n'entre pas dans la région.

Les critères 1 et 2 nous permettent de former des segments plus étendus et dès lors d'éliminer peut-être encore certaines instructions par les procédures de "Folding" et d'éliminations d'instructions redondantes.

Le critère 3 se justifie par le fait qu'il faut éviter de briser des liens LAR. N'oublions pas que des instructions modifiant la valeur d'une quantité sont susceptibles de se trouver dans le segment fictif.

<u>Exemple</u>

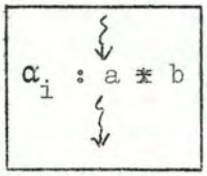


Supposons que la seule assignation à A dans la région ① - ② - ③ est A := 4. Dès lors, par les critères de déplacement en arrière, A := 4 est déplaçable et sera déposé dans le segment fictif. Si nous insérons le segment fictif dans le segment ⑥ , soit le lien LAR (A := 6 ⑥ , A * B ④)est brisé, soit le lien LAR (A := 4 ① , A * B ②)est brisé.

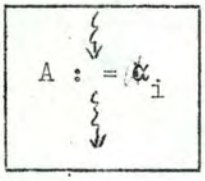
7.2. Remarques préliminaires

Avant l'insertion des instructions formant le segment fictif dans le programme, il convient d'analyser le segment fictif et la région R pour laquelle il a été généré, afin d'éviter les références inter-segments à des instructions intermédiaires.

Exemple



segment fictif

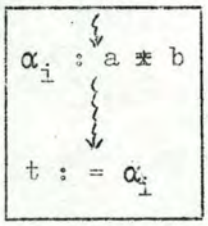


segment S figurant dans R

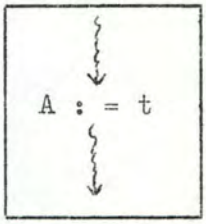
L'instruction α_i est référencée dans le segment S alors qu'elle n'est pas définie dans le segment S.

Pour toute instruction intermédiaire I définie dans le segment fictif et référencée dans un ou plusieurs segments de R, une variable temporaire t est générée. L'instruction t := I est placée dans le segment fictif et toute référence à l'instruction intermédiaire I dans un segment de R est remplacée par une référence à la variable générée t. Ceci est réalisé pour maintenir une propriété intéressante du texte intermédiaire, à savoir que toute instruction intermédiaire I n'est référencée que dans le segment où elle est définie. La plupart des procédures que nous avons examinées se basent sur cette propriété.

Exemple



segment fictif

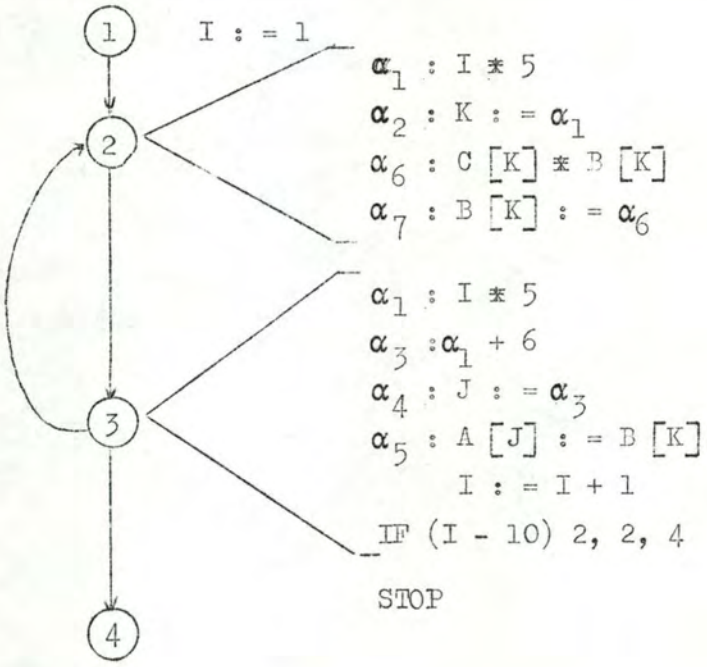


segment S figurant dans R

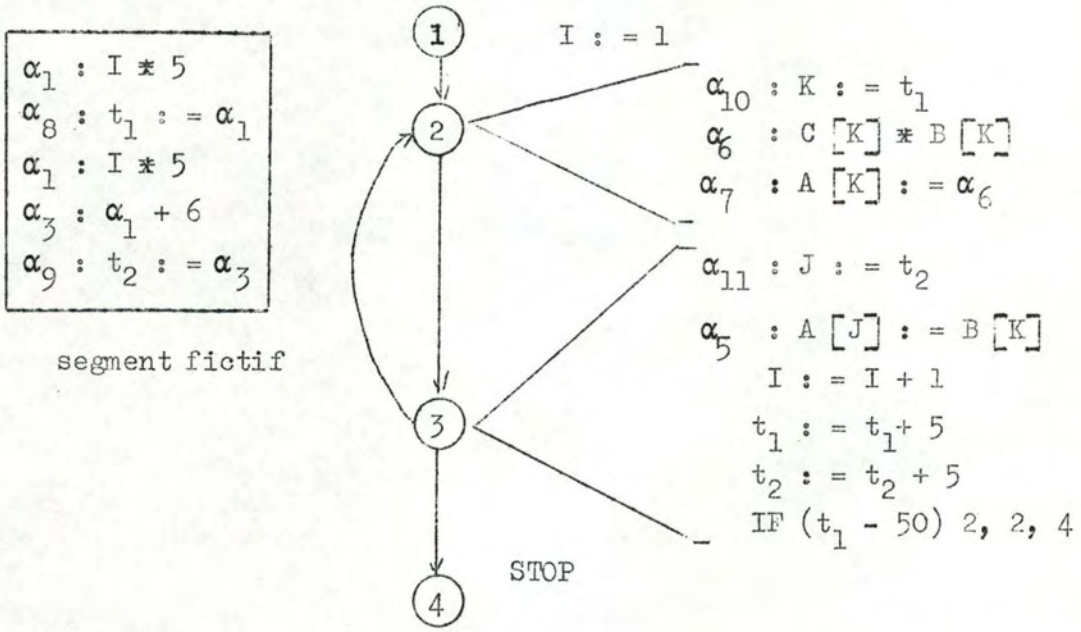
Parmi les instructions formant le segment fictif peuvent figurer des instructions redondantes ainsi que des instructions susceptibles d'être éliminées par la procédure "Folding" (instructions "Foldable"). C'est pour cette raison qu'avant d'insérer ces instructions dans le programme, il convient de les analyser par les procédures de "Folding" et d'élimination d'instructions redondantes. D'autre part, lorsque les instructions formant le segment fictif ont été insérées dans un segment S prédécesseur immédiat de la région, le nouveau segment S ainsi formé peut à nouveau contenir des instructions redondantes ainsi que des instructions "Foldable". On lui appliquera donc également les procédures de "Folding" et d'élimination des instructions redondantes.

Cela se fera lors de l'optimisation de la région couvrant la région optimisée ou lors de l'optimisation des segments ne faisant partie d'aucune région.

<u>Exemple</u> 1)



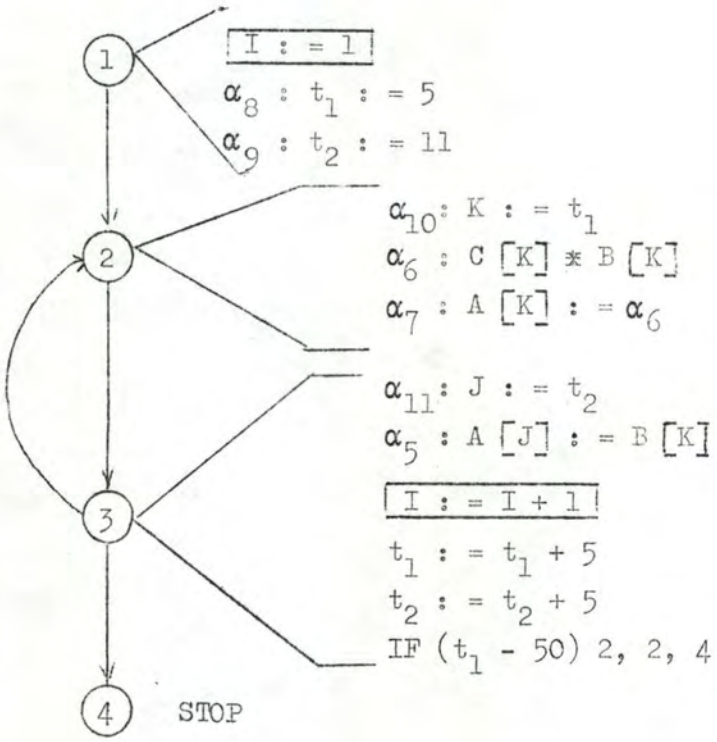
2) Contenu du segment fictif avant l'insertion et programme après optimisation de 2 - 3



Le contenu du segment fictif après "Folding" et élimination des instructions redondantes est le suivant :

α_1 : I * 5
α_8 : t_1 : = α_1
α_3 : α_1 + 6
α_9 : t_2 : = α_3

3) Après insertion du segment fictif dans le segment (1) et traitement du segment (1) étendu par les procédures de "Folding" et d'élimination des instructions redondantes, on a le programme suivant :



- Pour être complet, signalons que la procédure d'élimination des assignations mortes éliminera I := 1 et I := I + 1.

7.3. Procédure d'insertion

1) Les instructions formant le segment fictif sont analysées par les procédures de "Folding" et d'élimination des instructions redondantes.

2) Les références inter-segments à des instructions intermédiaires sont éliminées (voir Remarques préliminaires).

3) Les points d'entrée de la région sont déterminés.
Soit LPE = $\{ S_1, \ldots, S_m \}$ la liste de ces points d'entrée.

4) Les prédécesseurs immédiats de la région R sont recherchés. Ils sont les prédécesseurs immédiats des points d'entrée de R.
Soit LPI = $\{ P_1, \ldots, P_n \}$ la liste de ces prédécesseurs.

5) Dans la liste LPI, les prédécesseurs immédiats n'ayant qu'un seul successeur immédiat sont sélectionnés.
Soit LPISU = $\{ P_{i_1}, \ldots, P_{i_k} \}$ leur liste (LPISU $\subseteq$ LPI). Pour $i = i_1, \ldots, i_k$, les instructions formant le segment fictif sont insérées dans le segment P_i.

6) Sont ensuite sélectionnés, dans la liste LPI, les prédécesseurs immédiats jouissant des propriétés suivantes :

- ils possèdent plusieurs successeurs immédiats qui sont tous des points d'entrée de la région R;
- les instructions figurant dans le segment fictif n'affectent pas leur instruction de branchement.

Soit LPISPE = $\{P_{j_1}, P_{j_2}, \ldots, P_{j_e}\}$ la liste de ces prédécesseurs (LPISPE $\cap$ LPISU = $\emptyset$ et LPISPE $\subseteq$ LPI). Pour $j = j_1, \ldots, j_e$, les instructions formant le segment fictif sont insérées dans P_j.

7) LPISM = LPI $\setminus$ (LPISU $\cup$ LPISPE) = $\{P_{r_1}, \ldots, P_{r_h}\}$ $(k + e + h = n)$ est la liste des prédécesseurs immédiats de R possédant plusieurs successeurs immédiats qui ne sont pas tous des points d'entrée de R. L'insertion du segment fictif pour ce cas se déroule de la façon suivante.

7) a. La liste des segments érigés LSE est initialisée: LSE := $\emptyset$.

7) b. $i := r_1$.

7) c. La liste LSPE des successeurs immédiats de P_i qui sont des points d'entrée de R est créée. Soit LSPE = $\{S_{s_1}, \ldots, S_{s_{e(i)}}\}$

7) d. <u>for</u> $j := s_1$ <u>step</u> 1 <u>until</u> $s_{e(i)}$ <u>do</u>

<u>begin</u> <u>if</u> S_j a un prédécesseur immédiat figurant dans LSE

<u>then</u> - On change l'instruction de branchement de P_i de telle sorte que, depuis P_i, on puisse aller vers le prédécesseur immédiat de S_j figurant dans LSE.

- On met à jour la matrice booléenne C du graphe et les tables TDS, TI et TSQ.

<u>else</u> - On crée un nouveau segment érigé S qui est ajouté à la liste LSE. On change l'instruction de branchement de P_i de telle façon que le contrôle puisse aller de P_i à S et de S à S_j.

- On change en conséquence la matrice booléenne C du graphe et les tables TDS, TI et TSQ.

<u>end</u>

7) e. i : = i + 1

<u>if</u> i > r_n <u>then</u> <u>goto</u> END

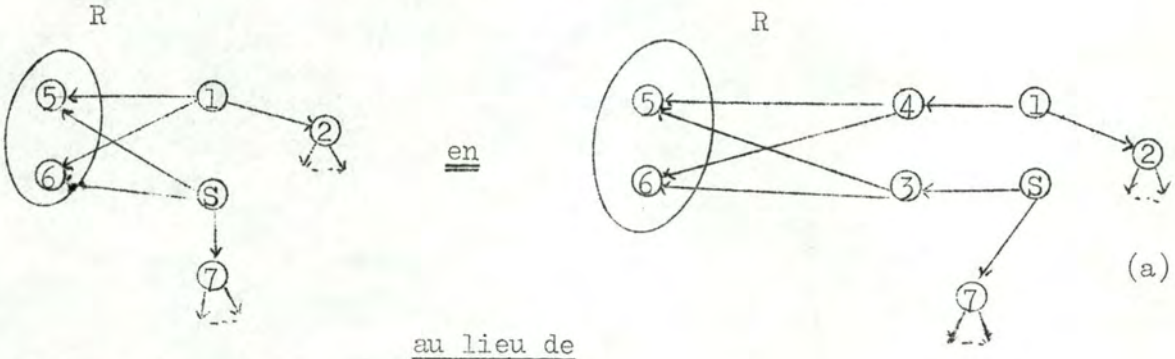
<u>else</u> <u>goto</u> (7).c

7) f. END : fin de la procédure

7.4. <u>Discussion</u>

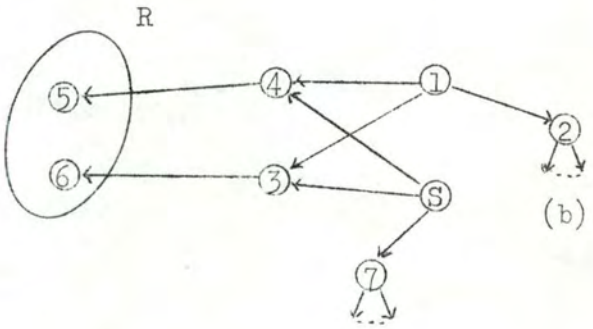
1) Le point d'entrée d'un programme ne fera jamais partie d'une région R car il est un sommet factice que l'on rajoute au graphe du programme et car il ne possède aucun prédécesseur. Ainsi ce sommet peut être un prédécesseur immédiat d'une région et est traité par la procédure d'insertion du segment fictif de la même façon que tout autre prédécesseur de la région.

2) On peut se demander pourquoi ne pas transformer



<u>au lieu de</u>

③ et ④ sont les segments érigés



Ceci suppose que nous recopions dans les segments érigés (ici ③ et ④) l'ensemble de la séquence d'évaluation de l'expression dont la valeur fera brancher dans un sens ou l'autre.

<u>Exemple</u> Supposons que dans (1) et (S) figurent les tests suivants :

IF ((A - B) - (C + D)) ⑤ , ⑥ , ②

IF ((A + B) - (C - D)) ⑤ , ⑥ , ⑦ .

Il faudra, pour réaliser (a) au lieu de (b), laisser les tests suivants :

dans 1, IF ((A - B) - (C + D)) ④ , ④ , ②

dans S, IF ((A + B) - (C - D)) ③ , ③ , ⑦.

Ceci ne demande pas beaucoup d'analyse, il suffit en effet de changer la table TDS.

Mais il faudra, dans ③ et ④ insérer les tests suivants :

IF ((A - B) - (C + D)) ⑤ , ⑥ , ②

IF ((A + B) - (C - D)) ⑤ , ⑥ , ⑦

ce qui demande bien plus d'analyse et amène, de plus, un gaspillage de la place mémoire.

L'option prise en 7.1 demande beaucoup moins d'analyse.

3) Si l'optimiseur réalise des déplacements en avant, les instructions déplacées se trouvent dans un autre segment fictif. Ce dernier doit être inséré sur tous les chemins sortant de la région. Un raisonnement analogue au précédent peut être tenu pour déterminer les critères d'insertion et la procédure.

CONCLUSIONS ET DEVELOPPEMENTS

Les techniques examinées dans ce travail sont essentiellement "langage indépendant" et "machine indépendante". Elles sont "langage indépendant" car elles sont applicables à une variété de langages algébriques. Elles sont "machine indépendante" car elles rendront un programme plus efficient sur une variété d'ordinateurs.

Nous pouvons remarquer également que l'utilisation d'un compilateur optimiseur est coûteuse. L'optimisation ne devient donc intéressante que lorsque le programme optimisé est un programme à réemploi fréquent. Il est en effet certain que le gain dû à l'optimisation est une fonction croissante du facteur de réemploi.
De ce fait, il n'est pas opportun d'optimiser un programme lors de sa mise au point. Il est préférable de ne le faire qu'après qu'un nombre suffisant de tests permettant de supposer le programme correct. Un autre facteur susceptible de retarder l'optimisation d'un programme est que le "debugging" est plus difficile, nous semble-t-il, sur le programme optimisé.
Il convient peut-être de nuancer la remarque relative à l'optimisation après une période de tests. En effet, le temps pris par l'optimiseur dépend du volume du programme * (grossièrement) et non de son temps d'exécution.
Cela signifie que l'on peut raisonnablement penser que l'on aura un temps d'optimisation croissant pour un volume croissant. Ainsi pour des programmes de petit volume, on pourra faire l'optimisation lors de chaque test.

L'intérêt du système présenté dans ce travail est qu'il permet une optimisation modulaire. Cela est d'autant plus intéressant que seulement un dixième ou un vingtième d'un programme possède une fréquence d'exécution élevée [KNUTH 71].

---

* Nous n'affirmons pas que le temps d'optimisation est le même pour des programmes ayant même volume.

L'optimisation de programme apporte une réponse aux problèmes de lisibilité et de maintenance des programmes.

Il nous semble que l'on peut répondre de façon affirmative aux questions posées à la fin de l'introduction. Il faudra sans doute nuancer la réponse à la troisième question, car il semble que le temps de compilation soit très important dans un système d'optimisation de programmes.

Il serait intéressant de faire une étude statistique sur la façon dont un langage est utilisé avant d'entamer l'étude d'un compilateur optimiseur pour ce langage [KNUTH 71]. En effet, dans un langage tel que Fortran, les expressions arithmétiques ont une longueur moyenne de deux opérandes. Il est par conséquent vain de créer un compilateur pouvant produire un code efficient pour des expressions arithmétiques plus complexes. Cette conclusion est aussi valable pour le déplacement d'instructions, la réduction des opérateurs ou toute autre technique.

Il reste beaucoup de problèmes que nous n'avons pas traités dans ce travail, par exemple :

1° l'allocation de registres;

2° le "linking" de sous-routines, (ceci se fait généralement au moyen de conventions standards qui ne sont pas toujours les plus optimales);

3° la généralisation des techniques du chapitre II au programme entier et non à ses segments;

4° l'optimisation de langages de gestion. Sans doute certaines techniques présentées dans ce travail restent valables, mais en raison de la structure particulière de ces langages, il est nécessaire de développer d'autres techniques (par exemple, l'élimination des opérations de conversion et de reconversion qui sont inutiles);

5° l'optimisation de langages conventionnels;

6° la factorisation des expressions arithmétiques (par exemple, A * B + A * C devenant A * (B + C));

7° l'étude théorique de l'optimisation de programmes.

8° etc...

—— § ——

ANNEXE 1

DEFINITION DE LA NOTION DE NIVEAU ETENDU

Définition de la notion d'intervalle

Etant donné un sommet h d'un graphe G possédant une seule entrée et une seule sortie, un intervalle I(h) est un sous-graphe maximal de G :

- possédant un seul point d'entrée, le sommet h;
- et tel que tous les circuits de ce sous-graphe passent par h.

h est appelé la tête de l'intervalle.

En choisissant un ensemble adéquat de sommets comme tête d'intervalle, un graphe peut être portionné en un ensemble d'intervalles.

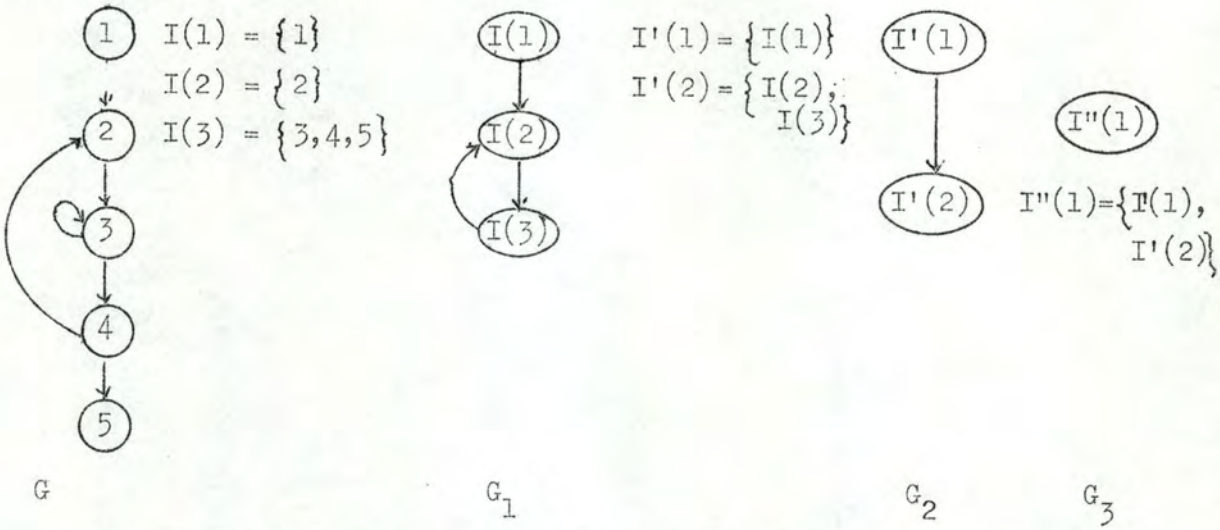
Exemple

I(1) = 1
I(2) = 2
I(3) = 3, 4, 5

Ayant décomposé le graphe G en intervalles, on peut en construire un nouveau, soit G_1, dont les sommets sont les intervalles de G. Les arcs de G_1 sont définis de la façon suivante: il existera dans G un arc reliant un intervalle I_i à un intervalle I_j s'il existe, dans G, un arc reliant un sommet de I_i à un sommet de I_j ($i \neq j$).

Le nouveau graphe ainsi obtenu peut à son tour être décomposé en intervalles et un nouveau graphe peut être formé au moyen de ces intervalles.

Exemple



Si le graphe obtenu en appliquant récursivement cette procédure se réduit à un seul sommet, nous disons que le graphe est <u>réductible</u>.

Pour plus d'information, nous renvoyons le lecteur à l'article d'Allen [ALLEN 70].

Le graphe de départ G est un graphe de degré 0, le graphe G_1 est de degré 1, G_2 de degré 2, etc...

Il s'agit de décomposer un graphe simplement connexe ayant un ou plusieurs sommets d'entrée et un ou plusieurs sommets de sortie, avec ou sans circuits, en niveaux étendus.
S'il présente un ou plusieurs sommets d'entrée, le graphe sera transformé de telle façon qu'il ne présente plus qu'un seul sommet d'entrée e_o, ceci en ajoutant le sommet e_o au graphe et les arcs (e_o, x_i) où x_i est un point d'entrée du graphe non transformé; de même s'il présente un ou plusieurs sommets de sortie, un sommet s_o lui sera ajouté ainsi que les arcs (x_j, s_o) où x_j est un sommet de sortie du graphe non transformé.

Le graphe transformé devra être réductible. S'il ne l'est pas, il faudra le transformer en un graphe réductible avant de lui appliquer la procédure de décomposition en niveaux étendus [MILLER 69].
Nous ne nous occuperons que des graphes réductibles. Cette décomposition partage les sommets du graphe transformé G'(X,U) en niveaux étendus de telle manière que

1) un sommet d'un niveau étendu peut posséder un prédécesseur dans les niveaux étendus qui suivent mais alors il doit exister dans le graphe G' un ou plusieurs chemins qui mène t du sommet vers son as endant (prédécesseur);

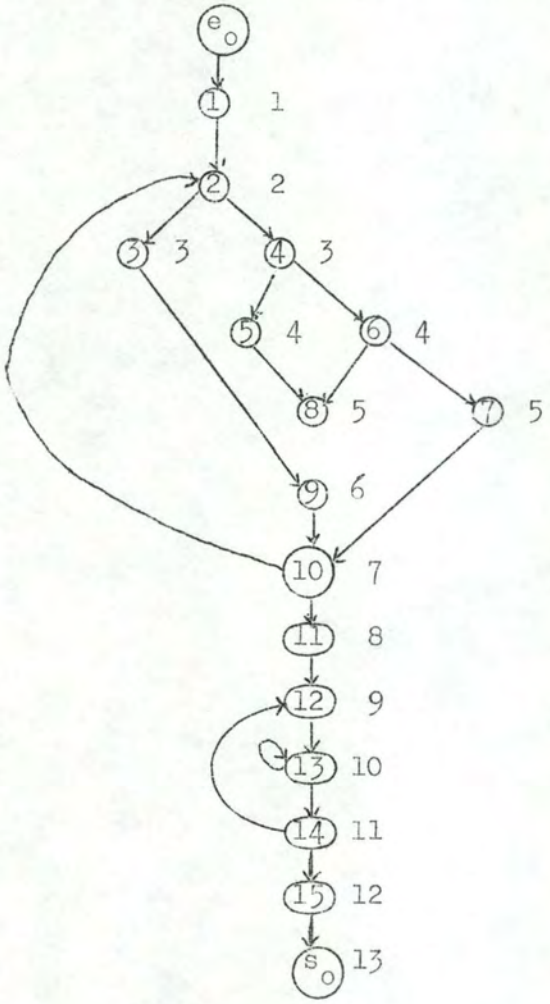
2) le sommet du premier niveau étendu est le seul et unique point d'entrée de G'; on lui affecte le niveau 0;

3) le sommet du dernier niveau étendu est le seul et unique point de sortie de G';

4) les sommets d'un même niveau étendu ne sont pas reliés entre eux par des chemins de longueur 1.

Le dernier critère (4) permet que deux sommets d'un même niveau étendu soient reliés entre eux par un circuit.

<u>Exemple</u> Le graphe suivant possède les propriétés requises.



Nous allons maintenant exposer une procédure montrant à suffisance pourquoi le graphe doit être réductible.

1) Affecter au sommet unique constituant le graphe réduit - supposons-le de degré N - le numéro de niveau étendu 0. Posons: $n := N$

2) $n := n - 1$

3) Créer la liste des intervalles du graphe de degré n :

$$L\,I = \{ I(h_i) \mid i = 1, \ldots, m \}$$

4) Ordonner L I de telle façon que $\forall\, i,j\ (1 \leq i < j \leq m)$, $n(I(h_i)) \leq n(I(h_j))$.

5) Pour tout intervalle $I(h_i)$ $(i = 1, \ldots, m) = \{h_i, x_1, \ldots, x_k\}$ de L I,

a) créer la liste PI des intervalles $I(h_p)$ $(p = 1, \ldots, m)$, $p \neq i$ prédécesseurs immédiats de $I(h_i)$ dans le graphe de degré n+1.

b) <u>si</u> $PI = \emptyset$ <u>alors</u> $n(h_i) := n(I(h_i))$

$n(I(h_i))$ = le numéro de niveau étendu du sommet représentant $I(h_i)$ dans le graphe de degré n+1. Aller en c)

<u>sinon</u> 1) créer à partir de PI la liste SPI des sommets $\in \bigcup_{I(h_i) \in PI} I(h_i)$

telle que d'un sommet $\in$ SPI, il existe un arc menant à h_i dans le graphe de degré n.

$$SPI = \{ x \mid x \in \bigcup_{I(h_i) \in PI} I(h_i) \text{ et } \exists\, (x, h_i) \in U_n \}$$

$$U_n = \{ \text{les arcs du graphe de degré } n \}$$

2) $n(h_i) = \max\ (n(x) \mid x \in SPI) + 1$

c) Si nous ignorons tous les arcs $(x, h_i) \in U_n$ et $x \in I(h_i)$, le sous-graphe du graphe de degré n formé par les sommets $I(h_i)$ est un graphe simplement connexe sans circuit, il est même un arbre. Ceci nous permet d'appliquer à ce sous-graphe la procédure de décomposition en niveaux d'un graphe simplement connexe sans circuits [FICHEFET 73].

6) <u>si</u> $n = 0$ <u>alors</u> stop <u>sinon</u> aller en 2.

Notation : $n(x)$ = le numéro de niveau étendu du sommet x dans le graphe de degré n.

A N N E X E 2

ANALYSE DE LA PROPAGATION DES DONNEES DANS UN PROGRAMME

Dans un programme divisé en segments d'instructions, nous désirons déterminer les définitions d'items de donnée qui affectent une utilisation d'un item de donnée. Plus particulièrement, nous désirons connaître l'ensemble des définitions d'items de donnée qui atteignent un segment d'instructions déterminé. Nous avons dû limiter nos recherches dans ce domaine à cause du volume et aussi à cause du temps. L'idée de base est extraite de l'article [ALLEN 72]. L'algorithme est basé sur un algorithme que nous avons développé à Swansea avec l'aide du professeur Cooper. Les démonstrations et le choix d'une heuristique optimale sont originaux.

a) ITEM DE DONNEE

Un item de donnée est un procédé de mémorisation, pendant une période de temps déterminée, d'une information élémentaire traitée dans le programme écrit dans un langage L.

Exemples - En FORTRAN IV :

1) les variables simples, quel que soit leur type R, I, Complexe, DP B
2) les éléments d'un tableau

sont des items de donnée.

En PL/1 dans une structure

```
DECLARE 1 USINE,
          2 ATELIER 1,
            3 NO ARTICLE FIXED,
            3 NAMECHIEF CHAR (20),
          2 ATELIER 2,
            3 NO ARTICLE FIXED,
            3 NAMECHIEF CHAR (20),
```

2 ATELIER 3,

3 NO ARTICLE FIXED,

3 N. MECHIEF CHAR (20),

les items de donnée sont

USINE.ATELIER 1.NO ARTICLE.

USINE.ATELIER 1.NAMECHIEF. et de même pour ATELIER 2 et ATELIER 3.

b) VALEUR D'UN ITEM DE DONNEE

La valeur d'un item de donnée à un instant donné est l'information élémentaire mémorisée à cet instant, ou en dernier lieu avant cet instant, par l'item de donnée.

Exemple

READ 3, (A,B)		
REAL X	→	valeur de l'item de donnée indéfinie X
X = A + B	→	" " " " X = A+B
READ 1, (X)	→	" " " " X: ce qu'il y a sur le support externe.
Y = X + C		
X = Y + D		
WRITE 2, (X)		

c) DEFINITION D'UN ITEM DE DONNEE

Une définition d'un item de donnée est toute instruction ou partie d'une instruction du programme P, écrit dans un langage L, qui, de par son exécution, modifie la valeur de l'item de donnée.

Exemples: A)
```
L 1 : begin real x; procedure p; x := x+1.; .......
          ; x := x + 5.; .......
          L 2 : begin real x; ......
                    ......; p; .....; x := 7.; .....
                end
      end
  end
```

a) x := x + 5 est une définition de l'item de donnée x déclaré dans le bloc L 1

b) ;p; est une définition de l'item de donnée x déclaré dans le bloc L 1

c) ;x := 7.; est une définition de l'item de donnée x déclaré dans le bloc L 2.

B)
```
     ||READ   901, (X, Y, Z)
     ||  :
901  ||FORMAT
```
est une définition des items de donnée X, Y et Z.

C) Dans :

```
begin  real  x; procedure p(x); value x; real x;
          x := x + 1;
          :
         ;p(x); .....
end
```
p(x) n'est pas une définition d'un item de donnée.

D) α_1: + ; A, B → définition de l'item α_1

α_2: * ; α_1,Z → définition de l'item α_2

α_3: :=; A, α_2→ définition de l'item A

Les α_i peuvent être des items de donnée vu qu'ils ont la possibilité de mémoriser temporairement de l'information élémentaire.

d) UTILISATION D'UN ITEM DE DONNEE

Une utilisation d'un item de donnée est toute instruction, ou partie d'une instruction, utilisant la valeur de l'item de donnée sans la modifier.

Exemples : a) b := f b = a then a + 5 else a + 10

a + 10 et a + 5 sont deux utilisations de l'item de donnée a.

b) l'exemple C du paragraphe c est une utilisation de l'item de donnée x

c) WRITE (90), (X,Y,Z) est une utilisation de trois items de donnée

d) α_2: *, α_1, Z est l'utilisation des items α_1 et Z.

e) DEFINITION RECURSIVE D'UN ITEM DE DONNEE

Une définition récursive d'un item de donnée est toute définition de l'item de donnée faisant une utilisation de l'item de donnée.

Exemple x := x + 4

```
begin real x; procedure p(x); real x;
                x := x + 4;
  ¦
  ¦
  ¦
;p(x); ........... end
```

p(x) est une définition récursive.

Plus tard, nous restreindrons la définition pour ne plus nous intéresser qu'aux définitions récursives du type x = f(x), où f est une fonction quelconque.

f) DEFINITION D'UN ITEM DE DONNEE AFFECTANT UNE UTILISATION D'UN ITEM DE DONNEE

On dit qu'une définition d'un item de donnée affecte une utilisation de cet item de donnée si, et seulement si, la valeur de l'item de donnée après la définition est la même que sa valeur lors de son utilisation.

Exemple Soit le programme FORTRAN IV :

```
1  X = A + B                    5  X = Z
2  Z = PI * R * * 2             6  A + PI + A
3  B = 2 * PI * R * * 2         7  B = X
4  A = B + X                       STOP
                                   END
```

La définition 1 de X affecte l'utilisation 4 de X mais n'affecte pas l'utilisation 7 de X.

Nous allons chercher à définir l'ensemble des définitions d'un item de donnée affectant une utilisation de cet item de donnée. S'il n'existait pas de branchement dans le programme source, ceci serait simple. En effet, on peut dire, dans ce cas, qu'une définition affecte une utilisation si, entre l'occurrence de la définition dans le programme P et l'occurrence de l'utilisation, les opérandes de l'instruction de définition ne sont pas à leur tour définis. Dans le cas où il n'y a pas de branchements, il n'y a jamais qu'une seule définition qui affecte une utilisation.

Lorsque nous permettons des branchements dans le programme P, plusieurs définitions peuvent affecter une utilisation. La définition f

ci-dessus, qui ne fait pas apparaître la notion de cheminement dans un graphe devra être transformée en une définition équivalente tenant compte de la notion de cheminement. Dès l rs, lorsque l'on veut démontrer qu'une définition affecte une utilisation, on est amené à trouver, dans le graphe, un chemin allant du segment contenant la définition vers le segment contenant l'utilisation et sur lequel aucun opérande de la définition n'est redéfini.

Nous sommes confrontés ici à deux problèmes :

1° trouver tous les chemins dans un graphe allant d'un sommet vers un autre;

2° trouver, parmi ces chemins, un chemin qui satisfasse aux conditions de la définition.

Ce genre d'algorithme, simple dans son énoncé, est peu efficace dans le cas présent.

Nous allons en présenter un autre qui n'est pas aussi intuitif. Nous introduirons auparavant quelques définitions nouvelles que nous classerons en deux catégories :

1° les définitions s'appuyant sur des informations locales,

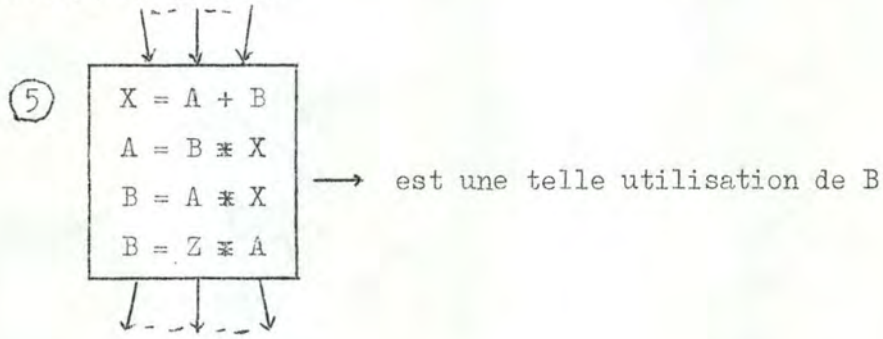
2° les définitions s'appuyant sur des informations globales.

(A) Définitions nécessitant une information locale

1) Utilisation susceptible d'être affectée localement

Une utilisation susceptible d'être affectée localement est une utilisation d'un item de donnée qui n'est pas précédé, dans le segment, par une définition de l'item de donnée.

Exemple



2) Définition localement accessible d'un segment

Une définition d'un item de donnée localement accessible d'un segment est la dernière définition de l'item de donnée dans le segment.

Exemple Dans l'ensemble précédent, B = Z * A, est une définition de B localement accessible.

Nous noterons Db_i l'ensemble des définitions localement accessibles de tous les items de donnée définis dans le segment de numéro i.

Exemple $Db_5 = \{ X = A + B,\ A = B * X,\ B = Z * A \}$

Nous noterons Pb_i l'ensemble des définitions du programme qui ne définissent aucun item de donnée défini dans le segment de numéro i.

(B) Définition nécessitant une information globale

1) Définition d appartenant à un segment i atteignant un segment k

Une définition d située dans un segment de numéro i atteint un segment de numéro k si et seulement si

1° $d \in Db_i$;

2° $\exists$ un chemin de i à k;

3° au moins un des chemins du segment i au segment k ne passe par aucun segment contenant une redéfinition de l'item de donnée défini par d dans le segment i.

2) Définition annulant une définition

Toute définition d_k d'un segment k annule une définition d atteignant le segment k, si d_k et d définissent le même item de donnée.

3) Définitions vivantes d'un item de donnée à la sortie d'un segment

Les définitions vivantes d'un item de donnée à la sortie d'un segment sont 1) soit la définition localement accessible de l'item de donnée dans le segment,

2) soit les définitions de l'item de donnée atteignant le segment, ceci à la seule condition que l'item de donnée ne soit pas redéfini dans le segment.

Nous désignerons par A_i l'ensemble des définitions vivantes à la sortie d'un segment i.
Dès lors, l'ensemble R_i des définitions atteignant un segment i est donné par la formule suivante :

$$\boxed{R_i = \bigcup_{p \in P_i} A_p} \qquad (1)$$

où $P_i = \{ p \}$ p est un numéro de segment et p prédécesseur immédiat de i

A_i sera alors donné par la formule suivante :

$$\boxed{A_i = (R_i \cap Pb_i) \cup Db_i} \qquad (2)$$

Connaissant Pb_i et Db_i pour chaque segment i, il est donc possible de calculer les A_i et R_i pour chaque segment i.
Nous remarquons que si le graphe associé au programme était simplement connexe et sans circuit, le calcul des A_i et R_i serait simple (décomposition d'un graphe en niveaux). Malheureusement, vu que la majorité des programmes présentent des circuits, il est nécessaire de trouver un algorithme permettant de calculer les A_i et R_i en sachant que l'on a commis une erreur et par conséquent de corriger l'erreur par la suite.

Le principe de l'algorithme que nous présentons ci-après se résume de la façon suivante. Lorsque, pour calculer R_i, on ne possède la valeur d'aucun A_p, $p \in P_2$, alors on suppose que $A_p = \emptyset$, $\forall\ p \in P_i$.
Dès que l'on possède une valeur possible pour un A_p, on la vérifie et on propage cette nouvelle information plus loin dans le graphe associé au programme.

Nous appellerons "devinette" le fait de supposer qu'un $A_p = \emptyset$.
Lorsque l'algorithme aura délivré une valeur (définitive ou pas) pour cet A_p, nous dirons que la devinette est annulée.

Nous supposons que le graphe associé au programme ne possède qu'un seul point d'entrée e_o et un seul point de sortie s_o. Si ce n'est pas le cas, il est facile de transformer le graphe pour qu'il possède la propriété.

Il suffit d'ajouter une entrée (sortie) fictive au graphe et de la relier aux entrées (sorties) réelles.

L'algorithme s'aidera d un marquage des sommets. Nous supposerons que ce marquage se fait par l'intermédiaire d'une fonction f qui, à un ensemble M de sommets marqués, fait correspondre un et un seul sommet marqué. Nous verrons qu'il n'est pas nécessaire de préciser explicitement la forme de f dans l'algorithme et qu'il suffit de supposer qu'elle existe. Nous montrerons néanmoins qu'il est possible de la choisir de telle façon que l'algorithme converge en un nombre minimum de pas; nous l'appellerons alors <u>heuristique optimale</u>.

Nous supposerons également que, pour chaque sommet X_k du graphe, sauf peut-être pour e_o et s_o, les quantités Db_k et Pb_k ont été calculées préalablement et sont mémorisées quelque part.
Nous supposerons qu'<u>avant de commencer l'algorithme</u> :

1) les A_k pour $X_k \in X$ ont été initialisés à $\emptyset$

2) R_{e_o} a été initialisé à D_{ext} (défini ci-dessous).

Les quantités Db_{e_o}, Pb_{e_o}, Db_{s_o}, Pb_{s_o} ont les valeurs suivantes :

$Db_{e_o} = D_{in}$ (défini ci-dessous), $Pb_{e_o} = Pb_{e_o}$, $Db_{s_o} = \emptyset$ et $Pb_{s_o} = D$ (défini ci-dessous).

<u>Définition de D, D_{in}, D_{ext}</u>

1) Le programme traité par l'algorithme peut être une sous-routine ou une procédure. Par conséquent, un certain ensemble D_{ext} de définitions extérieures au corps de la procédure ou sous-routine peuvent l'atteindre (plus particulièrement peuvent atteindre e_o). C'est la définition du langage source qui fixe en partie l'ensemble de ces définitions, de même que les lois d'affectation des utilisations des items de donnée dans le corps de la procédure ou sous-routine par ces définitions extérieures. Dans ce cas, il convient donc d'initialiser R_{e_o} à D_{ext} et d'enrichir l'ensemble D des définitions du programme au moyen de D_{ext} : $D = D_p \cup D_{ext}$, où D_p est l'ensemble des définitions figurant dans le corps de la sous-routine ou procédure. Dès lors, pour $X_k \in X \setminus \{e_o, s_o\}$ Pb_k devra être calculé en fonction de ce nouvel ensemble D; on a alors

$$Pb_k = Pb_k \cup D_{extcork}$$

où $D_{extcork}$ est l'ensemble des définitions extérieures qui ne définissent pas d'items de donnée définis dans le segment X_k.

2) Le langage source, de par sa définition, peut imposer une valeur initiale pour tous les items de donnée du programme ou seulement pour une partie d'entre eux. Ceci correspond à un ensemble de définitions qui serait inséré dans le sommet d'entrée e_o.
Soit D_{in} cet ensemble. Dans ce cas, D doit être modifié et $D = D_p \cup D_{in}$.
Tout comme en 1), les Pb_k doivent être calculés en fonction de ce nouvel ensemble D. Voici une formule de correction dans le cas où Pb_k est calculé en fonction de D_p uniquement

$$Pb_k = Pb_k \cup D_{in_{cor\ k}}$$

où $D_{in_{cork}}$ est l'ensemble des définitions d'initialisation que n'initialisent pas d'items de donnée définis dans x_k.

3) Dans le cas le plus général, $D = D_p \cup D_{ext} \cup D_{in}$, la formule de correction est alors, pour tout x_k,

$$Pb_k = Pb_k \cup (D_{ext_{cork}} \cup D_{in_{cork}})$$

4) D_{ext}, tout comme D_{in}, peut être vide, auquel cas $D = D_p$.

(voir notations en 9bis)

<u>Algorithme</u>

(1) - $A_{e_o} = (D_{ext} \cap Pb_{e_o}) \cup D_{in}$
- Marquer tout successeur immédiat de e_o

(2) <u>Si</u> $M = \emptyset$ <u>Alors</u> fin <u>Sinon</u> prenons $x_k = f(M)$

(3) - Démarquer x_k, c'est-à-dire pour $M = M \setminus \{x_k\}$.

(4) Calculer : $Px_k = \{p \mid x_p \in X$ et x_p prédécesseur immédiat de $x_k\}$,

$$R_k = \bigcup_{p \in Px_k} A_p,$$

$$A_k = (R_k \cap Pb_k) \cup Db_k.$$

Notations :

- A_k^{anc} est la valeur ancienne de A_k, c'est-à-dire la valeur la plus récente obtenue lors d'un pas antérieur de l'algorithme. Avant que l'algorithme ne commence, $A_k^{anc} = \emptyset$, $\forall$ k.

- $R_k^{anc} = \bigcup_{p \in Px_k} A_p^{anc}$

  où $Px_k = \{ p \mid x_p \in X$ et x_p immédiat prédécesseur de $x_k \}$

- $\subset$ est l'inclusion stricte

- $\subseteq$ est l'inclusion non stricte

- $\vee$ est le ou exclusif

B \ A	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Vrai

(5) <u>Si</u> $A_k^{anc} \neq A_k$, <u>Alors</u>

1) marquer tout suivant immédiat de x_k et poser :

$Sx_k = \{ x_j \mid x_j \in X$ suivant immédiat de $x_k \}$

$M = (M \cup Sx_k)$

2) $A_k^{anc} = A_k$

3) <u>Aller en</u> (2)

<u>Sinon</u> <u>Aller en</u> (2)

Nous appellerons <u>$n^{ème}$ pas</u> de l'algorithme la séquence de <u>paragraphes</u> (2), (3), (4), (5), retour à (2) ($n = 2, 3, \ldots, N < \infty$).

Nous appellerons <u>1er pas</u> de l'algorithme la séquence de <u>paragraphes</u> (1), (2), (3), (4), (5), (2).

Remarquons qu'à tout choix de la fonction f, il correspond un algorithme. Nous allons montrer que, quel que soit le choix de f, l'algorithme est fini, en ce sens qu'il se termine en un nombre fini de pas. Nous allons également montrer que si la fonction f est effective, c'est-à-dire telle que le choix du sommet x_k à traiter se fait dans un intervalle de temps fini, l'algorithme est également effectif, c'est-à-dire tel que son temps d'exécution est fini. A cet effet, nous supposerons que les ensembles D_{ext}, D_p, D_{in} sont finis. De ce fait, Db_i et Pb_i sont également des ensembles finis.

<u>Lemme 1</u> - Après chaque pas de l'algorithme et $\forall\, x_k \in X$, $\# A_k$ (1) est borné supérieurement par $\# D$ et $A_k \subseteq D$.

<u>Démonstration</u>

1) Après le premier pas, $\forall\, x_k \in X : \# A_k \leq \# D$

a) Après le premier pas, les seuls sommets $x_k \in X$ dont l'ensemble A_k est changé sont les sommets e_o et $x_k = f(M)$.

Ainsi, pour tout $x_j \in X \setminus \{ e_o, x_k \}$, $A_j = \emptyset$, d'où $A_j \subset D$ et $\# A_j \leq \# D < \infty$.

---

(1) $\# A_k$ désigne le cardinal de l'ensemble A_k.

b) $A_{e_o} = (D_{ext} \cap Pb_{e_o}) \cup D_{in}$

$(D_{ext} \subset D \text{ et } Pb_{e_o} \subseteq D) \Rightarrow ((Pb_{e_o} \cap D_{ext}) \subseteq D)$

$((Pb_{e_o} \cap D_{ext}) \subseteq D \text{ et } D_{in} \subset D) \Rightarrow (A_{e_o} \subseteq D)$

$(A_{e_o} \subseteq D) \Rightarrow \# A_{e_o} \leqslant \# D.$

c) - $R_k = \bigcup_{p \in Px_k} A_p = A_{e_o} \cup (\bigcup_{p \in Px_k \setminus \{e_o\}} A_p) = A_{e_o} (x_k = f(M))$

Puisque $A_p = \emptyset$ pour $p \in Px_k \setminus \{e_o\}$, il résulte de b) que $R_k \subseteq D$ et que $\# R_k \leqslant \# D$.

- $A_k = (R_k \cap Pb_k) \cup Db_k$

On a $R_k \subseteq D$ et $Pb_k \subseteq D$, d'où $(R_k \cap Pb_k) \subseteq D$.

Donc, vu que $Db_k \subseteq D$, nous aurons que $A_k \subseteq D$ et $\# A_k \leqslant \# D$.

Nous pouvons donc affirmer qu'après le premier pas,

$$\forall x_k \in X : \begin{cases} A_k \subseteq D \\ \text{et} \\ \# A_k \leqslant \# D \end{cases}$$

2) Supposons qu'après chaque pas 1, 2, ..., n de l'algorithme, $(\forall x_k \in X;\ A_k \subseteq D \text{ et } \# A_k \leqslant \# D)$ et démontrons qu'il en est encore ainsi après le pas n+1.

- Soit $f(M) = x_k$ au $(n+1)^{\text{ème}}$ pas

- Après le $(n+1)^{\text{ème}}$ pas de l'algorithme, il existe un ensemble $X\,\emptyset^{n+1}$ de sommets $x_j \in X$ tels que $A_j \neq \emptyset$. Pour $x_j \notin X\,\emptyset^{n+1}$ $A_j = \emptyset$.

- A la fin du $(n+1)^{\text{ème}}$ pas, le seul sommet $x_j \in X$ qui a vu la valeur de son A_j changer est x_k. Pour les autres sommets x_j, A_j a gardé sa valeur précédemment acquise au cours des pas 1, 2, ..., n. Par l'hypothèse de récurrence, nous pouvons donc affirmer que pour $x_j \in X \setminus \{x_k\}$, $A_j \subseteq D$ et $\# A_j \leqslant \# D < \infty$. Par l'hypothèse de récurrence, nous pouvons affirmer également, $A_k^{anc'}$ étant la valeur de A_k^{anc} avant l'affectation de A_k à A_k^{anc}, que $A_k^{anc'} \subseteq D$ et $\# A_k^{anc'} \leqslant \# D$.

- Par conséquent

a) si $k \in Px_k$ <u>Alors</u> $\left(R_k = A_k^{anc'} \cup \left(\bigcup_{p \in Px_k \setminus \{k\}} A_p\right)\right) \subseteq D$,

si $k \notin Px_k$ <u>Alors</u> $\left(R_k = \bigcup_{p \in Px_k} A_p\right) \subseteq D$,

donc $R_k \subseteq D$ et $\# R_k \leq \# D$.

b) $(A_k = (R_k \cap Pb_k) \cup Db_k) \subseteq D$

d'où $\# A_k \leq \# D$.

<u>Note</u>.- La relation $A_k^{anc} \subseteq D$ après chaque pas de l'algorithme est également vraie. $\Rightarrow \# A_k^{anc} \leq \# D < \infty$.

De 1) et de 2), nous pouvons donc conclure qu'après un pas quelconque et $\# A_k \leq \# D < \infty$ et $A_k \subseteq D$, $\forall x_k \in X$.

<u>Lemme 2</u> - <u>Si</u> $\mathcal{A} = \{A_{p_1}, \ldots, A_{p_k}\}$ et $\mathcal{A}' = \{A'_{p_1}, \ldots, A'_{p_k}\}$ $(k=1,2,\ldots,n)$ sont deux ensembles d'ensembles de définitions vivantes à la sortie des segments $x_{p_1}, \ldots, x_{p_k}$ qui sont tous des prédécesseurs immédiats d'un même sommet x_j, <u>si</u> $\mathcal{A}$ et $\mathcal{A}'$ sont tels que $A_{p_i} \subset A'_{p_i}$, $\forall p_i \in \{p_1, \ldots, p_k\}$

et <u>si</u> $P \times C_j = \{p_i \mid i=1,\ldots,k\}$

$$R'_j = \left(\bigcup_{p \in Px_j \setminus PxC_j} A_p\right) \cup \left(\bigcup_{p \in PxC_j} A'_p\right) \text{ et } R_j = \left(\bigcup_{p \in Px_j \setminus PxC_j} A_p\right) \cup \left(\bigcup_{p \in PxC_j} A_p\right)$$

$$A_j = (R_j \cap Pb_j) \cup Db_j$$

$$A'_j = (R'_j \cap Pb_j) \cup Db_j$$

<u>Alors</u> (1°) $R_j \subset R'_j$

(2°) $(A_j \subset A'_j) \vee (A_j = A'_j)$ ($\vee$ ou exclusif)

<u>Démonstration</u> (1°) trivial

(2°) il découle de (1°) que

$$((R_j \cap Pb_j) \subset (R'_j \cap Pb_j)) \vee ((R_j \cap Pb_j) = (R'_j \cap Pb_j))$$

c'est-à-dire $(A_j \subset A'_j) \vee (A_j = A'_j)$

<u>Lemme 3</u> - Dans n'importe quel pas de l'algorithme pour lequel $x_k = f(M)$, la relation entre A^{anc}_k et A_k est

$$(A^{anc}_k \subset A_k) \vee (A^{anc}_k = A_k)$$

au moment du test du paragraphe (5).

<u>Démonstration</u> - 1) Le lemme est vrai au moment du test (5) dans le premier pas.

a) Les seuls sommets traités au premier pas sont e_o et $x_k = f(M)$ qui est un successeur immédiat de e_o.

b) $A^{anc}_{e_o} = \emptyset$

$A_{e_o} = (D_{ext} \cap Pb_{e_o}) \cup D_{in}$.

<u>Si</u> $D_{ext} = D_{in} = \emptyset$ <u>Alors</u> $A_{e_o} = \emptyset = A^{anc}_{e_o}$

<u>Si</u> $D_{ext} \neq \emptyset = D_{in}$ <u>Alors</u> $(A_{e_o} = (D_{ext} \cap Pb_{e_o})) \supset A^{anc}_{e_o}$

<u>Si</u> $D_{ext} = \emptyset \neq D_{in}$ <u>Alors</u> $(A_{e_o} = D_{in}) \supset A^{anc}_{e_o}$

$$\Rightarrow (A^{anc}_{e_o} \subset A_{e_o}) \vee (A^{anc}_{e_o} = A_{e_o})$$

c) $\forall x_j \in X \setminus \{x_k\}$ $A^{anc}_j = \emptyset = A_j$ et $A^{anc}_k = \emptyset$, d'où :

- $R^{anc}_k = \bigcup_{p \in Px_k} A^{anc}_k = \bigcup_{p \in Px_k} \emptyset = \emptyset$ même si $k \in Px_k$

- $R_k = A_{e_o} \cup (\bigcup_{p \in Px_k \setminus \{e_o\}} A_p) = A_{e_o}$ si $k \notin Px_k$.

- $R_k = A_{e_o} \cup (\bigcup_{p \in Px_k \setminus \{k, e_o\}} A_p) \cup A^{anc}_k = A_{e_o}$ si $k \in Px_k$

d'où $R^{anc}_k \subseteq R_k$ et

$$((R^{anc}_k \cap Pb_k) \subset (R_k \cap Pb_k)) \vee ((R^{anc}_k \cap Pb_k) = (R_k \cap Pb_k)).$$

(Pb_k pourrait être vide)

Par conséquent $(A^{anc}_k \subset A_k) \vee (A^{anc}_k = A_k)$

2) Supposons qu'au moment du test en (5) dans les pas 1,2,.. .., n, le lemme 3 soit vrai. Supposons que l'on ait mémorisé A^{anc}_j

pour les x_j traités au cours des pas 1,2,...,n juste avant l'affectation de A_j à A_j^{anc}, et que, pour un x_j traité plusieurs fois au cours des pas 1,2,.., n, on n'ait gardé que le A_j^{anc} le plus récent avant l'affectation. Cet A_j^{anc} sera écrit $A^{anc'}$ par la suite(1).

Démontrons que le lemme est encore vrai au pas (n+1).

- Soit $f(M) = x_k$ au $(n+1)^{ème}$ pas.
- Le fait que x_k soit marqué signifie que, pour un certain nombre de x_p, $p \in Px_k$, on a $A_p^{anc'} \neq A_p$ au moment du test (5) dans les pas 1,2,...,n de l'algorithme. Par l'hypothèse de récurrence, la relation entre ces $A_p^{anc'}$ et A_p est $A_p^{anc'} \subset A_p$. En effet, si l'on avait eu $A_p^{anc'} = A_p = A_p^{anc}$, on n'aurait pas marqué x_k à partir de ce sommet x_p, ce qui est contraire à l'hypothèse faite sur x_p.

  Soit $P \times C_k = \{ p \mid p \in Px_k \text{ et } A_p^{anc'} \neq A_p \}$ ou sous une forme équivalente, par l'hypothèse de récurrence,

  $$P \times C_k = \{ p \mid p \in Px_k \text{ et } A_p^{anc'} \subset A_p \}$$

  Soient alors $-\mathcal{A} = \{ A_p^{anc'} \mid p \in P \times C_k \}$

  $$-\mathcal{A}' = \{ A_p \mid p \in P \times C_k \}$$

$\mathcal{A}$ et $\mathcal{A}'$ satisfont aux conditions du lemme 2.

Soit $k \in Px_k \setminus P \times C_k$. Nous avons :

$$R_k^{anc} = \Big(\bigcup_{p \in PxC_k} A_k^{anc'} \Big) \cup \Big(\bigcup_{p \in (Px_k \setminus PxC_k) \setminus \{k\})} A_p \Big) \cup A_k$$

puisque $A_k^{anc'} = A_k = A_k^{anc}$ pour $k \notin P \times C_k$, d'où

$$R_k^{anc} = \Big(\bigcup_{p \in PxC_k} A_k^{anc'} \Big) \cup \Big(\bigcup_{p \in Px_k \setminus PxC_k} A_p \Big)$$

De même,

$$R_k = \Big(\bigcup_{p \in PxC_k} A_p \Big) \cup \Big(\bigcup_{p \in ((Px_k \setminus PxC_k) \setminus \{k\})} A_p \Big) \cup A_k^{anc'}$$

$$= \Big(\bigcup_{p \in PxC_k} A_p \Big) \cup \Big(\bigcup_{p \in (Px_k \setminus PxC_k)} A_p \Big)$$

---

(1) Si x_o est traité deux ou plusieurs fois au cours des pas 1,2,...,n et si le A_j^{anc} de cet x_j lors du test (5) devient égal à A_j, alors on posera $A_j^{anc'} = A_j^{anc} = A_j$.

Soit $k \in P \times C_k$. Nous pouvons écrire que

$$R^{anc}_k = \left(\bigcup_{p \in PxC_k \setminus \{k\}} A^{anc'}_p \right) \cup A^{anc}_k \cup \left(\bigcup_{p \in Px_k \setminus PxC_k} A_p \right)$$

Ici, pour $k \in PxC_k$, $A^{anc'}_k \subset A^{anc}_k = A_k$ et $A^{anc'}_k \in \mathcal{A}$

$$R^{anc}_k \subset \left(\bigcup_{p \in PxC_k \setminus \{k\}} A^{anc'}_p \right) \cup \left(\bigcup_{p \in Px_k \setminus PxC_k \cup \{k\}} A_p \right)$$

$$R_k = \left(\bigcup_{p \in PxC_k \setminus \{k\}} A_p \right) \cup A_k \cup \left(\bigcup_{p \in Px_k \setminus PxC_k} A_p \right)$$

ici, $k \in P \times C_k$ $\quad A_k \in \mathcal{A}'$

$$= \left(\bigcup_{p \in PxC_k} A_p \right) \cup \left(\bigcup_{p \in Px_k \setminus PxC_k} A_p \right)$$

Les ensembles R^{anc}_k, R_k, $A^{anc}_k = (R^{anc}_k \cap Pb_k) \cup Db_k$ et $A_k = (R_k \cap Pb_k) \cup Db_k$ satisfont donc, $\forall\, k \in Px_k$, aux conditions du lemme 2, d'où :

$$R^{anc}_k \subset R_k \quad \text{et} \quad (A^{anc}_k \subset A_k) \vee (A^{anc}_k = A_k)$$

<u>Corollaire</u> - Dans n'importe quel pas de l'algorithme pour lequel $x_k = f(M)$, nous avons

$$(\# A^{anc}_k < \# A_k) \vee (A^{anc}_k = A_k)$$

Il en résulte que le test $A^{anc}_k \neq A_k$ du paragraphe (5) de l'algorithme peut encore s'écrire $\# A^{anc}_k < \# A_k$.

<u>Théorème 1</u> - L'algorithme est fini, quelle que soit la fonction de choix f.

<u>Démonstration</u>

A chaque pas de l'algorithme, juste avant le test (2) (si $M \neq \emptyset$), nous pouvons associer un couple de nombres entiers (m, S) au graphe représentant le programme :

$$m = \# M$$

$$S = \sum_{x_k \in X} (\# A_k)$$

Soit $x_i = f(M)$ le sommet sélectionné au cours d'un pas donné. Posons:

$$S^{anc} = \sum_{x_k \in X \setminus \{x_i\}} (\# A_k) + (\# A_i^{anc'})$$

(A) m et S sont des nombres entiers finis. m l'est certainement puisque $M \subseteq X$ et que X est fini par hypothèse. De plus, en vertu du lemme 1, nous savons que, $\forall x_k \in X$: $\# A_k \leq \# D < \infty$; S est donc également fini.

(B) S est borné supérieurement par une constante $S_{max} \leq n * (\# D)$ puisqu'en vertu du lemme 1, $\# A_k \leq \# D$, $\forall x_k \in X$.
S ne peut pas décroître strictement d'un pas à un autre, c'est-à-dire $S^{anc} \leq S$.

Soit n un pas quelconque et $f(M) = x_k$ au cours de ce pas. Nous avons:

$$S = \sum_{x_i \in X} (\# A_i) = \sum_{x_i \in X \setminus \{x_k\}} (\# A_i) + (\# A_k)$$

$$S^{anc} = \sum_{x_i \in X \setminus \{x_k\}} (\# A_i) + (\# A_k^{anc'})$$

Le corollaire du lemme 3 nous affirme qu'au moment du test (5) au cours d'un pas quelconque de l'algorithme,
$(\# A_k^{anc} < \# A_k) \vee (A_k^{anc} = A_k)$
ou encore que $(\# A_k^{anc} < \# A_k) \vee (\# A_k^{anc} = \# A_k)$.
Or, à ce moment-là, par définition de A_k^{anc} et $A_k^{anc'}$, on a que $A_k^{anc'} \subseteq A_k^{anc}$, d'où $S^{anc} \leq S$.

(C) Pour démontrer maintenant que l'algorithme se termine en un nombre fini de pas, quelle que soit la fonction de choix f, nous démontrerons que S atteint une borne supérieure après un nombre fini de pas de l'algorithme. Une fois que S a atteint sa borne supérieure S_{max}, nous pouvons affirmer, par le corollaire du lemme 3, qu'à chaque pas de l'algorithme, si $f(M) = x_k$, $A_k^{anc} = A_k$ au moment du test (5) et donc qu'à partir de ce moment-là m diminue de une unité à chaque pas de l'algorithme.
Comme m est un entier fini, cela implique qu'à partir du moment où S a atteint sa borne supérieure, l'algorithme se termine en un nombre fini de pas ($m = o \Longleftrightarrow M = \emptyset$).

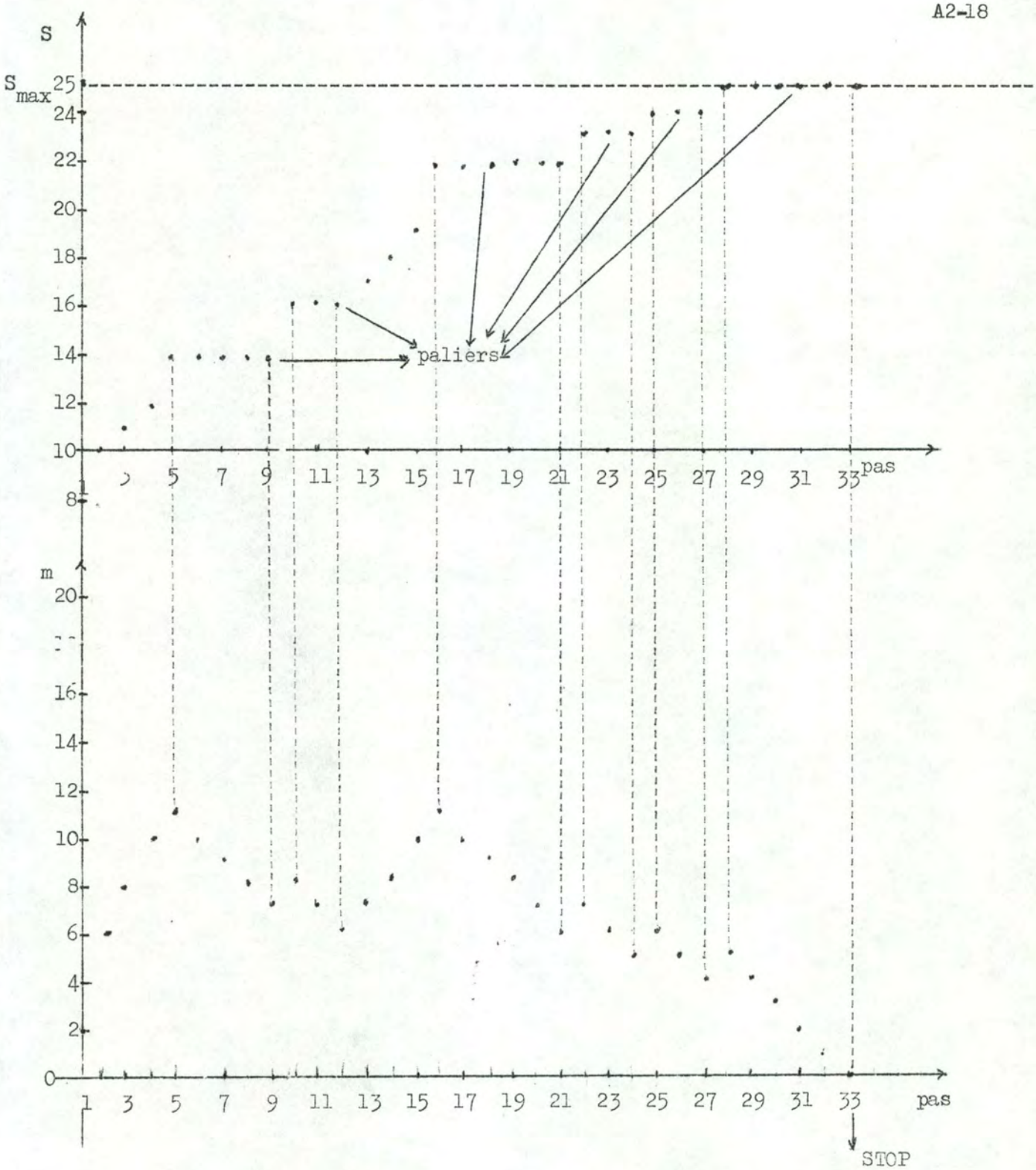
(D) Deux cas sont possibles: ou bien S croît strictement, ou bien la croissance de S est non stricte.

a) S croît strictement. Dans ce cas, à chaque pas de l'algorithme $S^{anc} < S$. Comme, entre $S = 0$ et $S = S_{max}$, il n'y a qu'un nombre fini de valeurs entières possibles, S atteindra S_{max} après un nombre fini de pas de l'algorithme et l'algorithme se terminera en un nombre fini de pas ((D)).

b) S ne croît pas strictement à chaque étape, c'est-à-dire $(S^{anc} < S) \vee (S^{anc} = S)$. Pour démontrer que S atteint sa borne après un nombre fini de pas de l'algorithme, il faut démontrer que l'état $S^{anc} = S$, que nous appellerons un palier, ne peut pas se prolonger indéfiniment d'un pas à l'autre de l'algorithme.

Nous allons nous aider des graphiques suivants pour S et m afin d'illustrer le raisonnement.

(voir page suivante)



Entre S = 0 et S = S_{max}, il n'y a qu'un nombre fini P de paliers puisque S_{max} est fini: S ne peut donc occuper qu'un nombre fini p ($p \leq P$) de paliers. Lorsque S arrive à un palier k, il n'a pu occuper qu'un nombre fini de

paliers entre le palier $S = 0$ et $S = S_k$ (k). Lorsque S occupe un palier k de valeur S_k, nous allons voir que deux possibilités peuvent se présenter: ou bien l'algorithme se termine en un nombre fini de pas ou bien S quitte le palier après un nombre fini de pas vers un nouveau palier ou vers S_{max} sa borne supérieure. Par (A) $m = m_k < \infty$ lorsque $S = S_k$ est fini. Or, si la valeur de S ne varie pas d'un pas à un autre de l'algorithme, nous pouvons dire, en vertu du lemme 3, que m diminue d'une unité à chaque pas. Dès lors, il n'existe que deux possibilités :

a) $m_k \rightarrow 0$ après un nombre fini de pas de l'algorithme.
Dans ce cas, $S_k = S_{max}$ et l'algorithme se termine en un nombre fini de pas puisque S ne peut rester sur chaque palier que pendant un nombre fini de pas et que S n'a pu occuper qu'un nombre fini de paliers.

b) S quitte le palier avant que m_k n'ait atteint la valeur zéro.
Dans ce cas S n'est resté sur le palier qu'un nombre fini de pas et de plus, S quitte le palier pour accroître sa valeur. S se dirige donc vers un palier supérieur et on peut recommencer le raisonnement. Cela chève de démontrer le théorème.

Théorème 2 - L'algorithme est effectif si la fonction de choix f est effective.

Démonstration

Comme l'algorithme n'implique que des opérations d'union et d'intersection d'ensembles finis (opérations qui sont effectives), l'algorithme est effectif.

Plus précisément :

- le temps t_1 passé du paragraphe (1) est fini
- le temps t_2 passé du paragraphe (2) est fini pas hypothèse
- le temps t_3 passé du paragraphe (3) est également fini puisque $\# M < \infty$.
- de même, les temps t_4 et t_5 passés aux paragraphes (4) et (5) respectivement sont finis.

Comme l'algorithme est lui-même fini, il se déroule en un nombre

fini N de pas, d'où

$$t_{total} = t_1 + N\,(t_2 + t_3 + t_4 + t_5) < \infty$$

Nous allons maintenant définir une fonction de choix f telle que le nombre de pas de l'algorithme soit minimal pour un graphe déterminé. Nous démontrerons que l'algorithme est correct dans une recherche ultérieure. Correct signifie que l'ensemble des définitions qui atteignent un sommet x_i du graphe à l'exécution est un sous-ensemble de R_i calculé à la compilation durant la phase d'optimisation.

RECHERCHE D'UNE HEURISTIQUE OPTIMALE

Nous appelons recherche d'une heuristique optimale la recherche de la définition - indépendante de la structure du graphe associé à un programme particulier - d'une fonction de choix f, définition qui minimise le nombre de pas de l'algorithme pour tout le programme donné.

Cette recherche peut être envisagée de deux manières différentes(*)

1° Soit P un programme donné quelconque et G (X, τ) son graphe associé. Nous pourrions déterminer tout d'abord l'ensemble de toutes les fonctions de choix f(M) possibles et définies indépendamment d'un graphe particulier. Soit $\mathcal{F} = \{ f_1, \ldots, f_m \}$ (**) cet ensemble. Nous pourrions ensuite déterminer l'ensemble $\mathcal{A} = \{ A_1, \ldots, A_m \}$ des algorithmes correspondants et les appliquer successivement au programme P. Ceci nous donnerait un ensemble $\mathcal{N} = \{ n_1, \ldots, n_m \}$ de nombres entiers finis qui sont respectivement les nombres de pas des A_i appliqués à P.

La fonction f_k choisie serait celle qui correspond à $n_k = \min(n_i \mid n_i \in \mathcal{N})$. Bien qu'elle ait l'avantage de bien formuler le problème de notre recherche, cette approche est très peu efficace. La longueur de sa mise en oeuvre dans le temps risque de décourager la plupart d'entre nous; de plus, rien ne nous assure qu'elle est effective $(n < \infty)$.

(*) Il en existe peut-être d'autres.
(**) $\mathcal{F}$ pourrait éventuellement être infini dénombrable.

2° Ayant défini la notion d'heuristique optimale, nous pourrions dégager des critères généraux auxquels la fonction devra satisfaire. Nous pourrions alors tenter de démontrer que toute fonction f satisfaisant à ces critères est une heuristique optimale (théorème d'optimalité). Nous devrions démontrer par la même occasion que notre système S de critères est complet. Cela signifie qu'il n'existe pas d'autres critères pouvant être ajoutés à S ou mis à la place de certains critères de S pour fournir un autre système S' dans lequel toute fonction f' satisfaisant aux critères de S' serait une heuristique plus optimale que f (une heuristique optimale satisfaisant aux critères de S).

Nous adopterons ici la deuxième approche mais nous nous limiterons à déterminer des critères généraux pour f. La démonstration des deux théorèmes, si elle existe, est reportée au futur.

Définitions

(A) Jusqu'ici, nous n'avons défini que de façon vague les notions de "devinette" et de "devinette" annulée. Nous développerons maintenant des définitions plus précises.

(B) Rappel
- $Px_k = \{ p \mid x_p \in X$ et x_p prédécesseur immédiat de $x_k \}$
- x_k désignera toujours dans la suite du texte le sommet choisi au cours du pas que l'on envisage.
- Un sommet x_p est dit accessible depuis l'entrée e_o s'il existe au moins un chemin allant de e_o vers x_p.

(C) "Devinette"
- Nous dirons qu'au cours d'un pas, l'algorithme est amené à faire une "devinette" si l'une des trois éventualités suivantes est vérifiée (ou toute combinaison de ces dernières).

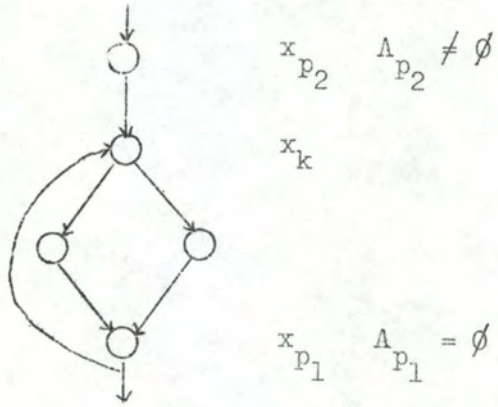
1) Pour le calcul de $R_k = \bigcup_{p \in Px_k} A_p$ au cours du pas, il existe au moins un $A_p = \emptyset$, x_p étant accessible depuis l'entrée e_o, et il existe un chemin allant de x_k à x_p.
2) Il n'existe aucun x_p possédant les propriétés du 1). Mais il existe un x_p, avec $p \in Px_k$, pour lequel $A_p^{anc} \neq A_p$

au test (5) la dernière fois que l'algorithme a sélectionné x_p par l'intermédiaire de f; de plus, il existe un chemin allant de x_k vers x_p.

3) Il existe au moins un sommet $x_j \in M$ parmi les prédécesseurs de x_k (immédiats ou non).

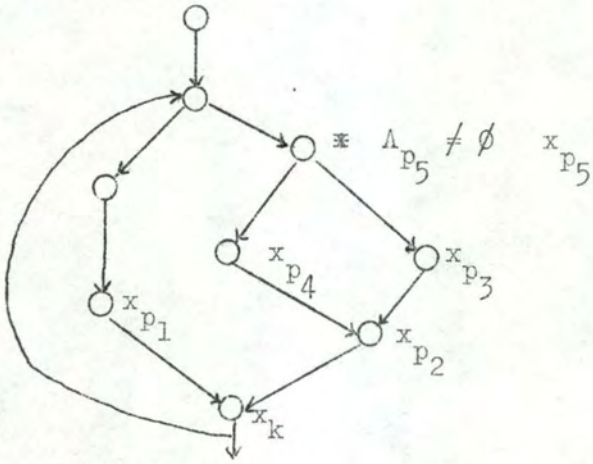
<u>Exemple de "devinette"</u>

(1)



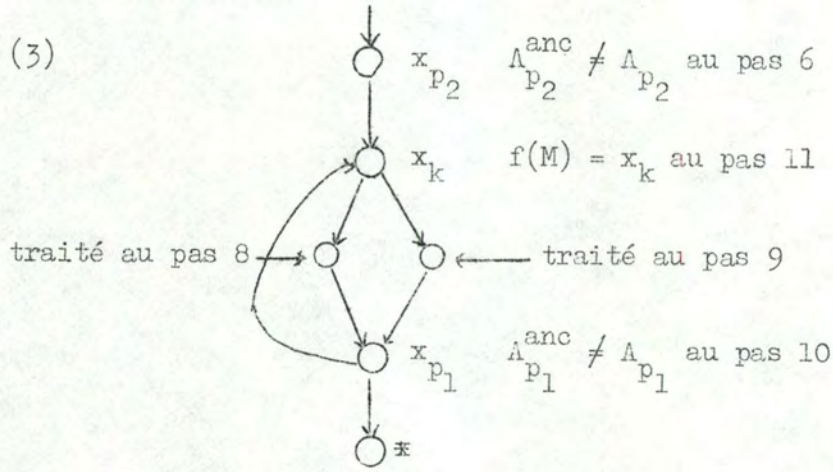
En choisissant x_k l'algorithme fait une "devinette".

(2) * indiquera qu'un sommet est marqué.

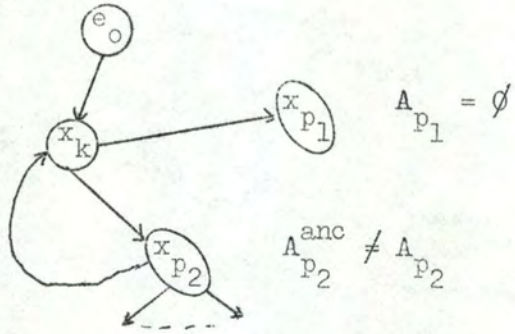


En choisissant x_k, l'algorithme fait une devinette. C'est ce genre de devinettes qu'il faudra éviter.

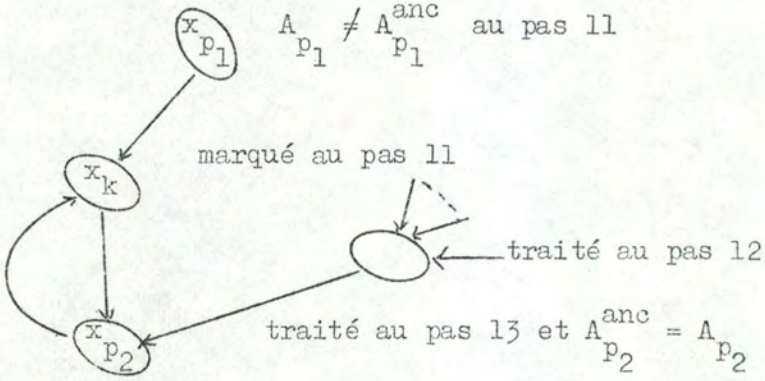
(3)



<u>Contre-exemples</u>



x_{p_1} est un sommet inaccessible. En choisissant x_k, on ne fait pas de "devinette".



Si x_k est choisi au pas 14, l'algorithme ne fait pas une "devinette".

(D) Une séquence de pas de l'algorithme est une suite ordonnée dans le temps $p_1, p_2, \ldots, p_n$ $(n \leq N < \infty)$ de pas de l'algorithme, telle que, t_j étant le temps pris pour l'achèvement du pas, p_j et t_{in} l'instant auquel le pas p_1 a débuté, le pas p_k $(1 < k \leq n)$ ne débute qu'à l'instant $t_{in} + \sum_{i=1}^{k-1} t_i$, et telle que seul p_1 peut être le premier pas (①, ②, ③, ④, ⑤, ②).

(E) "Devinette" temporairement annulée par un pas

Nous dirons qu'une "devinette" faite lors du traitement d'un sommet x_j est temporairement annulée par le pas actuel de l'algorithme si le pas traite un sommet x_k prédécesseur immédiat de x_j et si $A_k^{anc} \neq A_k$ au moment du test (5).

Une conséquence immédiate de cette définition est que le sommet x_j figurera après le pas dans la liste M des sommets marqués. De la même façon, si un sommet x_j pour lequel une "devinette" avait été faite lors d'un pas antérieur réapparaît dans la liste M, nous pouvons dire que la devinette a été temporairement annulée par le pas qui a fait réapparaître x_j dans M.

(F) Définition équivalente à la définition (E)

Nous dirons qu'une "devinette" faite lors du traitement d'un sommet x_j est temporairement annulée par le pas actuel si ce pas fait entrer x_j dans l'ensemble M des sommets marqués.

(G) "Devinette" définitivement annulée

Nous dirons qu'une "devinette" faite lors du traitement d'un sommet x_j est définitivement annulée si une séquence quelconque de pas suivant le pas au cours duquel la "devinette" a été faite, a éliminé de l'ensemble M tout prédécesseur, immédiat ou non, de x_j, y compris x_j lui-même.

Nous pouvons conclure de cette définition que, dès lors qu'une devinette faite lors du traitement d'un sommet x_j est définitivement annulée, $A_j^{anc} = A_j$ est le résultat final pour le sommet x_j, puisque x_j ne peut plus jamais figurer dans la liste M.

(H) "Devinette" dans un état non défini

Nous dirons qu'une "devinette" faite lors d'un pas p_1 est dans un état non défini au cours d'un pas ultérieur p_n si elle n'est annulée ni

définitivement ni temporairement après la séquence de pas $(p_1, \ldots, p_n)$ (p_n pouvant être le même que p_1).

(I) Vérification d'une "devinette" temporairement annulée

Nous dirons qu'au pas actuel, il y a vérification d'une "devinette" temporairement annulée par un pas antérieur pour le sommet x_j si, lors du pas actuel, x_j est choisi et traité.

De la définition même de la vérification d'une "devinette" temporairement annulée, nous pouvons conclure qu'une vérification augmente le nombre de pas de l'algorithme d'au moins une unité. De plus, une "devinette" temporairement annulée sera toujours vérifiée, d'après sa définition même et celle de la vérification. D'autre part, une vérification d'une "devinette" temporairement annulée peut annuler définitivement la "devinette". Lorsque cela se présente, le nombre de pas de l'algorithme augmente d'une unité seulement; dans le cas contraire, il augmentera sûrement de plus d'une unité. En effet, le fait que la "devinette" n'a pas été définitivement annulée par la vérification implique nécessairement que certains prédécesseurs du sommet x_j pour lequel on a vérifié la "devinette" sont entrés dans la liste M; ceci a pour conséquence soit de réannuler temporairement la "devinette" si le sommet x_j est son propre prédécesseur, soit de placer la "devinette" dans un état non défini.

Théorème 3 - Toute "devinette" dans un état non défini au cours d'un pas deviendra ultérieurement soit définitivement soit temporairement annulée.

Démonstration - Soit $\mathcal{D}$ une devinette située dans un état non défini par un sommet x_j au cours du pas p_1. Il existe donc une séquence de pas $(p'_1, \ldots, p'_n)$, avec $p'_n = p_1$ à l'issue de laquelle $\mathcal{D}$ n'a été :

- ni annulée définitivement: dans ce cas, il existe au moins un sommet $x_p \in M$ et un chemin du graphe allant de x_p à x_j,
- ni annulée temporairement: dans ce cas, il n'existe aucun $x_p \in PX_j$ pour lequel $A_p^{anc} \neq A_p$ dans le test (5) de l'algorithme au cours des pas $p'_1, \ldots, p'_n$, ce qui signifie que x_j n'est pas marqué.

Prenons un sommet x_p (on sait qu'il existe) marqué qui est un prédécesseur de x_j et appliquons-lui l'algorithme. Les possibilités suivantes peuvent se présenter :

a) x_p est un prédécesseur immédiat de x_j et $A_p^{anc} \neq A_p$ au moment du test (5) : la "devinette" devient temporairement annulée.

b) x_p est le seul prédécesseur immédiat de x_j qui soit marqué et $A_p^{anc} = A_p$: la "devinette" devient définitivement annulée.

c) x_p est un prédécesseur marqué de x_j mais n'est pas le seul. Si $x_p \in PX_j$ et $A_p^{anc} \neq A_p$, on retombe dans le cas a).
Si $x_p \notin PX_j$, alors, quel que soit le résultat du test de (5), on a toujours une "devinette" non définie et on peut réitérer le processus jusqu'à ce que l'on tombe sur un des cas a), b) et c). Ce processus n'est pas infini puisque l'algorithme est fini et s'arrête lorsque $M = \emptyset$ auquel cas la "devinette" est définitivement annulée.

Le théorème suivant est une conséquence directe du théorème 3 :

Théorème 4 - Il existe toujours une séquence de pas consécutive au pas au cours duquel une "devinette" est faite, telle que la devinette devienne :

1) soit temporairement annulée;

2) soit définitivement annulée.

Le théorème et les définitions précédentes ont une conséquence fort importante pour le choix de notre premier critère d'optimalité.
En effet, ils nous disent que toute "devinette" peut augmenter le nombre de pas de l'algorithme et qu'elle le fera sûrement si elle n'est pas définitivement annulée. Dès lors, notre premier critère consiste à minimiser le nombre de "devinettes".

Nous avons vu également qu'une vérification d'une "devinette" temporairement annulée conduit l'algorithme à cheminer dans le graphe associé au programme pour propager l'information. Or, le cheminement dans le graphe de l'algorithme entraîne automatiquement une hausse du nombre de pas. Il est dès lors intéressant de vérifier le nombre maximum de "devinettes" en un nombre restreint de cheminements (2me critère).

Nous savons également qu'une vérification est provoquée automatiquement par une annulation temporaire d'une "devinette". L'annulation temporaire d'une "devinette" n'est rien d'autre que l'apport de nouvelles informations à l'entrée d'un sommet. Il est donc important dans certains cas, de posséder

au plus vite la nouvelle information à la sortie du sommet pour pouvoir la propager plus loin encore dans le graphe et travailler avec une information aussi complète que possible. C'est ce que fait la vérification d'une "devinette". Ce sera notre troisième critère.

Si nous ne tenions pas compte de ce troisième critère, il est fort probable que la vérification apporterait une information nouvelle tellement puissante qu'il faudrait cheminer longuement dans le graphe pour la propager.

On remarquera que ces trois critères sont souvent contradictoires et qu'il est dès lors nécessaire de trouver un juste milieu.

Résumons maintenant les critères que nous venons de dégager.

Critère d'optimalité pour la fonction de choix f

Lors du déroulement de l'algorithme, il faut que la fonction f choisisse un sommet x_k qui amène l'algorithme:

1) à faire un nombre minimum de "devinettes",

2) à vérifier un nombre maximum de "devinettes" temporairement annulées en un nombre minimum de cheminements dans le graphe,

3) à disposer au plus vite de la vérification d'une "devinette" annulée temporairement.

Les démonstrations du théorème d'optimalité et du théorème de complétude nous obligent à trouver un outil mathématique adéquat pour formaliser ces trois critères. Cet outil est loin d'être trivial et nous ne poursuivrons pas cette recherche ici.

Remarques sur la correction de l'algorithme

Nous ne démontrerons pas la correction de notre algorithme mais nous donnerons un énoncé de ce que nous comprenons par algorithme "correct".

Lorsqu'il se termine, l'algorithme délivre l'ensemble des définitions d'items de donnée atteignant un segment d'instructions. R_k est l'ensemble des définitions d'items de donnée qui atteignent un segment x_k.

Lors de l'exécution du programme, des définitions d'items de donnée sont rencontrées et exécutées. Chaque fois qu'une définition d d'un item de

donnée v est rencontrée à l'exécution, nous la plaçons dans une table T à condition toutefois qu'il n'existe pas déjà une définition de v dans T. S'il exis e déjà une telle défini ion dans T, nous la rem laçons par d.

L'algorithme sera dit "correct" si, quel que soit le chemin $u = e_o, \ldots, x_n$ suivi dans le graphe G, lors de l'exécution du programme, T est un sous-ensemble de R_n.

La démonstration de la correction de notre algorithme pourrait se faire au moyen du process calculus défini par Peter Bachmann [BACHMANN 72] et adapté aux besoins de la cause.

Débouchés de l'algorithme

Connaissant l'ensemble des définitions d'items de donnée atteignant un segment, nous pouvons déterminer les définitions qui affectent une utilisation déterminée dans un segment; nous pouvons également déterminer les définitions qui sont "mortes" et qui sont "vivantes" (au sens du chapitre 6).

ANNEXE 3

REMARQUES SUR LE LANGAGE ALGOL-JONQUILLE-74

Le langage Algol-Jonquille-74 est un langage similaire au langage Algol 60 pour la partie algébrique et Cobol pour les structures. Quelques concepts nouveaux y sont introduits.

1) Template list : est une liste de types que peut posséder une quantité.

Exemple (INTEGER, REAL) VAL signifie que la variable VAL peut à la fois être du type entier ou réel.

2) EXCEPT (Template list) Q signifie que la quantité Q peut posséder tous les types permis par le langage, excepté ceux qui se trouvent dans la "Template list".

3) Les types possibles sont les types permis par Algol 60, Fortran 4, plus les types CHARACTER, BINARY, REFERENCE, LABELMEMORY et STRUCTURE.

Le type CHARACTER est identique au type alphabétique de Cobol.

Le type BINARY permet de manipuler des chaînes de caractères binaires.

Le type REFERENCE est identique au type REFERENCE du langage "Algol Bastard" de Gries [GRIES 71].

Le type LABELMEMORY permet de manipuler des étiquettes du programme.

Le type STRUCTURE est identique aux structures de données dans le langage Cobol. Les structures sont pourtant dynamiques.

4) L'instruction :

JUMPWITHRETURN L_1, IMPLICIT Labelmemory;

s'interprète de la façon suivante. Il s'agit de brancher à l'instruction Algol-Jonquille-74 possédant l'étiquette L_1 et de charger, dans une variable de type LABELMEMORY, l'étiquette de l'instruction suivante.

Si l'instruction suivante n'est pas étiquetée, alors une étiquette doit être générée par le compilateur.

Il y a des extensions possibles pour cette instruction.

5) L'instruction :

JUMPBACK Labelmemory;

s'interpr te de la façon suivante Il s'agit de brancher à l'instruction Algol-Jonquille-74 possédant l'étiquette mémorisée par la variable Labelmemory de type LABELMEMORY.

6) Il devra exister dans ce langage des primitives pouvant tester le type d'une quantité.

BIBLIOGRAPHIE

1) [ALLEN 69] Allen, F.E., Program optimization, Annual Review in Automatic Programming, vol. 5, Pergamon, New York, 1969, pp. 239-307.

2) [ALLEN 72] Allen, F.E., A basis for program optimization, Information Processing 71, North Holland Publishing Company, 1972, pp. 385-390.

3) [ALLEN 70] Allen, F.E., Control flow analysis, Proceedings of a symposium on compiler optimization, Sigplan notices, July 1970, pp. 1-19.

4) [AHO 70] Aho, A.V., Ravi Sethi, Ullman, J.D., A formal approach to Code optimization, Proceedings of a symposium on compiler optimization, Sigplan notices, July 1970, pp. 86-100.

5) [BALLARD 72] Ballard, A., Tsichritzis, D., Transformations of programs, Information processing 71, North-Holland Publishing Company, 1972, pp. 414-418.

6) [BUSAM 69] Busam, V.A., Englund, D.E., Optimization of expressions in Fortran, CACM, vol. 12, 1969, pp. 666-674.

7) [BACHMANN 72] Bachmann, P., A contribution to the problem of the optimization of programs, Information processing 71, North-Holland Publishing Company, 1972, pp. 397-401.

8) [BRUEUER 69] Brueuer, Generation of optimal code for expressions via factorization, CACM, vol. 12, 1969, pp. 333-340.

9) [COCKE 70] Cocke, J., Schwartz, J.T., Programming languages and their compilers, Courant Institute of Mathematical Sciences, New York University, Avril 1970, pp. 306-523.

10) [COCKE 70] Cocke, J., Global common sub-expression elimination, Proceedings of a symposium on compiler optimization, Sigplan notices, Juillet 1970, pp. 20-24.

11) [COOPER 68] Cooper, D.C., Some transformations and standard forms of graphs, with applications to computer programs, Machine intelligence, Oliver and Boyd, Edinburgh and London, 1968, pp. 21-32.

12) [COOPER 66] Cooper, D.C., Reduction of programs to a standard form by graph transformation, International Seminar on Graph Theory and its Applications, Rome, Juillet 1966, Dunod, Paris.

13) [DREYFUS 70] Dreyfus, M., Fortran IV, Dunod, Paris, 1970.

14) /ELSON/ Elson, M., Rake, S.T., Code-generation technique for large language compilers, I.B.M. Systems Journal, vol. 9, n° 3, pp. 166-188.

15) /FICHEFET 73/ Fichefet, J., Théorie des graphes, Notes de cours, F.N.D.P., Namur, 1973.

16) /FLOYD 61/ Floyd,R.,An algorithm for coding efficient arithmetic operations, CACM, vol. 4, Janvier 1961, pp. 42-51.

17) /GRIES 71/ Gries,D.,Compiler Construction for digital computers, Wiley, 1971.

18) /GEAR 65/ Gear, C.W., High speed compilation of efficient object code, CACM, vol. 8, Août 1965, pp. 483-488.

19) /HOPKINS 72/ Hopkins, M., An optimizing compiler design, Information Processing 71, North-Holland Publishing Company, 1972, pp. 391-396.

20) /KNUTH 71/ Knuth, D.E., An empirical study of Fortran programs, Stanford University, Computer Science Department, Report CS-186.

21) /KNUTH 68/ Knuth, D.E., The art of computer programming, vol. 1, Addison-Wesley, World Student Series Edition,1972.

22) /HOPGOOD 69/ Hopgood, F.R.A., Compiling Techniques, Mac Donald, London, American Elsevier, New York, 1969.

23) /NIEVERGLET 65/ Nieverglet, J., On the automatic simplification of computer programs, CACM, vol. 8, n° 6, Juin 1965, pp. 366-370.

24) /LM 69/ Lowry, E.S., Medlock, C.W., Object code optimization, CACM, vol. 12, n° 1, Janvier 1969, pp. 13-22.

25) /ERSHOV 66/ Ershov, A.P., ALPHA - an automatic programming system of high efficiency, JACM, vol. 13, n° 1, Janvier 1966, pp. 17-24.

26) /ERSHOV 67/ Ershov, A.P. (ed.), Economy and Allocation of the memory in the ALPHA-translator, ALPHA - an automatic programming system, Nauka, Novosibirsk, 1967.

27) /Mc KEEMAN 65/ Mc Keeman, W.M., Peep hole Optimization, CACM, vol. 8, 1965, pp. 443-444.

28) /RUTISHAUSER 67/ Rutishauser, H., Description of Algol 60, Handbook for Automatic Computation, vol. 1, Part a, Springer-Verlag, Berlin, 1967.

29) [DOWSING] Dowsing, R.D., Systems programming course, Notes de cours, University College of Swansea, Swansea.

30) [SCHNECK 72] Schneck, P.B., Angel, E., A Fortran to Fortran optimising Compiler, The Computer Journal, vol. 16, n° 4, pp. 322-330.

31) [MILLER 69] Miller, R., Cocke, J., Some analysis techniques for optimizing Computer programs, Proceedings Second International Conference of Systems Sciences, Hawaï, Janvier 1969.

—— § ——